CHAPTER 5

METHODOLOGIES

5.1 INTRODUCTION

The Data Cleaning process Data Cleansing: An Efficient Technique Based on Finger Printing and Edit Distance identifies dirty records by finding those records that represent a real world entity already represented by another record. After that, it cleans the dirty records using the following steps (i) Reducing them to get a unified whole free of omitted parts, (ii) uniting them with a single individual identity and (iii) Keeping only one copy of records that are absolute duplicates (Kukich 1992).

5.1.1 Data Quality

Data Quality Management (DQM) is the pipeline process that checks the data for required values, valid data types, and valid codes. It can also be configured DQM to correct the data by providing default values, formatting numbers and dates, and adding new codes. Data quality management, along with name hygiene and standardization and address hygiene and standardization, is designed to optimize and enhance data quality. This data quality preparation is an essential step in entity resolution, because it increases the confidence in the resulting resolved entities and detected relationships. To apply data quality management to the data loaded into the system, configure data quality management rules (or DQM rules). DQM rules can perform a variety of repair, clean up, and standardization functions on
incoming identity data values, such as properly formatting numbers, identifying and correcting clerical or transposition errors, and identifying and correcting intentional inaccuracies introduced by those intent on trying to conceal their identities. The product comes pre-configured with several DQM rules by UMF segment that handle the most typical data quality issues for that UMF segment. But it can configure additional DQM rules, as needed. Before do so, however, it should be familiar with the original quality of the data and the ETL (extract, transform, and load) process that was used to transform the identity data into UMF. After knowing what further data enhancement is necessary, it can be selected the right DQM rules, functions, and values to apply to each type of identity data that needs further data quality optimization. Warehouse Builder architecture for data quality, it can create the own data quality mappings. Warehouse Builder provides functionality that enable to ensures data quality. During transformation of the source data, it can also use the following operators to ensure data quality:

- Match-Merge Operator
- Name and Address Operator
- Example of using a DQM rule

For example, the date format for the system is DD/MM/YYYY. But in several of data sources, the date values are formatted as MM-DD-YYYY. It can be added the DQM rule 204 to the <NUMBER> UMF segment, configuring it to fix all incoming dates formatted as MM-DD-YYYY to the date format of DD/MM/YYYY.

Good quality data means that all master data is complete, consistent, accurate, time-stamped and industry standards-based. By improving the quality of data, trading partners reduce costs, improve productivity and accelerate speed to market. Good quality data is foundational
to collaborative commerce and global data synchronization. GS1 GDSN calls
for data quality programmes that are sustainable and focused on the long
term: experience has shown time and again that business benefits come not
from enacting short-term curative data cleansing actions, but only from
having good quality data from the start. GS1, along with AIM, CIES, ECR
Europe, FMI, GCI and GMA have developed a comprehensive best practice
guide for the improvement of data quality for global data quality called the
Data Quality. Quality data is crucial to decision-making and planning. The
aim of building a data warehouse is to have an integrated, single source of
data that can be used to make business decisions. Since the data is usually
sourced from a number of disparate systems, it is important to ensure that the
data is standardized and cleansed before loading into the data warehouse.
Warehouse Builder provides functionality that enables to effectively manage
data quality by assessing, transforming, and monitoring the data. The benefits
of using Warehouse Builder for data management are as follows:

Provides an end-to-end data quality solution. Enables to include data
quality and data profiling as an integral part of the data integration process.
Stores metadata regarding the quality of the data alongside with data. Both the
data and the metadata are stored in the design repository. Automatically
generates the mappings that can use to correct data. These mappings are based
on the business rules that choose to apply on the data. And decision that
makes on how to correct data. Phases in the Data Quality Life Cycle. Ensuring
data quality involves the following phases:

- Quality Assessment
- Quality Design
- Quality Transformation
- Quality Monitoring.
Quality Assessment

In the quality assessment phase, determining the quality of the source data. The first step in this phase is to load the source data, which could be stored in different sources, into Warehouse Builder. It can import metadata and data from both Oracle and non-Oracle sources. After loading the source data, use data profiling to assess its quality. Data profiling is the process of uncovering data anomalies, inconsistencies, and redundancies by analyzing the content, structure, and relationships within the data. The analysis and data discovery techniques form the basis for data monitoring.

Quality Design

The quality design phase consists designing the quality processes. It can be specified the legal data within a data object or legal relationships between data objects using data rules.
Quality Transformation

The quality transformation phase consists of running the correction mappings that are used to correct the source data.

Quality Monitoring

Data monitoring is the process of examining the data over time and alerting the user when the data violates any business rules that are set. Data profiling is the first step for any organization to improve information quality and provide better decisions. It is a robust data analysis method available in Warehouse Builder that it can use to discover and measure defects in the data before it can start working with it. Because of its integration with the ETL features in Warehouse Builder and other data quality features, such as data rules and built-in cleansing algorithms, it can also generate data cleansing and schema correction. This enables to automatically correct any inconsistencies, redundancies, and inaccuracies in both the data and metadata.

Data profiling enables to discover many important things about the data. Some common findings include the following:

- A domain of valid product codes
- A range of product discounts
- Columns that hold the pattern of an e-mail address
- A one-to-many relationship between columns
- Anomalies and outliers within columns

Relations between tables even if they are not documented in the database Warehouse Builder enables to automatically create correction
mappings based on the results of data profiling. On top of these automated corrections that make use of the underlying

Data Rules Data rules are definitions for valid data values and relationships that can be created in Warehouse Builder. They determine legal data within a table or legal relationships between tables. Data rules help ensure data quality. They can be applied to tables, views, dimensions, cubes, materialized views, and external tables. Data rules are used in many situations including data profiling, data and schema cleansing, and data auditing. The metadata for a data rule is stored in the repository. To use a data rule, apply the data rule to a data object.

Quality Monitoring Quality monitoring builds on the initial data profiling and data quality initiatives. It enables to monitor the quality of the data over time. It can be defined the business rules to which the data should adhere. To monitor data using Warehouse Builder need to create data auditors. Data auditors ensure that the data complies with the business rules defined. It can define the business rules that the data should adhere to using a feature called data rules.

5.1.2 Data Cleansing

Data cleansing is the process of amending or removing data in a database that is incorrect, incomplete, improperly formatted, or duplicated. An organization in a data-intensive field like banking, insurance, retailing, telecommunications, or transportation might use a data scrubbing tool to systematically examine data for flaws by using rules, algorithms, and look-up tables. Typically, a database scrubbing tool includes programs that are capable of correcting a number of specific type of mistakes, such as adding missing zip codes or finding duplicate records. Using a data scrubbing tool
can save a database administrator a significant amount of time and can be less
costly than fixing errors manually. Data cleansing, data cleaning or data
scrubbing is the process of detecting and correcting (or removing) corrupt or
inaccurate records from a record set, table, or database. Used mainly in
databases, the term refers to identifying incomplete, incorrect, inaccurate,
irrelevant, etc. parts of the data and then replacing, modifying, or deleting this
dirty data. After cleansing, a data set will be consistent with other similar data
sets in the system. The inconsistencies detected or removed may have been
originally caused by user entry errors, by corruption in transmission or
storage, or by different data dictionary definitions of similar entities in
different stores. Data cleansing differs from data validation in that validation
almost invariably means data is rejected from the system at entry and is
performed at entry time, rather than on batches of data. The actual process of
data cleansing may involve removing typographical errors or validating and
correcting values against a known list of entities. The validation may be strict
(such as rejecting any address that does not have a valid postal code) or fuzzy
(such as correcting records that partially match existing, known
records). Some data cleansing solutions will clean data by cross checking with
a validated data set. Also data enhancement, where data is made more
complete by adding related information, is a common data cleansing practice.
For example, appending addresses with phone numbers related to that
address. Data cleansing may also involve activities like, harmonization of
data, and standardization of data. For example, harmonization of short codes
(St, rd etc.) to actual words (street, road). Standardization of data is a means
of changing a reference data set to a new standard, ex, use of standard codes.

Accuracy of decision support analysis on data warehouses is
essential because it influences important business decisions. But, errors such
as spelling mistakes and incompatible principles exist in the data that is
obtained at the data warehouse from external resources. Therefore, considerable quantity of time and funds are expended for data cleansing or in other words, for the data recognition and correction task. The proposed work will remove duplicate records, near-duplicate records, misspellings, missing values and illegal values from a dataset and the cleansing process is performed on dataset preferably in the order (i) Preprocessing: Only useful data are extracted from raw dataset. (ii) Duplicate record detection and deletion: Exactly matched records and nearly matched records are detected and removed by applying fingerprint and edit distance based techniques simultaneously. (iii) Correcting misspelling errors: Misspellings in textual formats are identified and corrected based on edit distance technique. (iv) Finding missing value errors and illegal value errors: Missing and illegal characters or numerical values present in chosen fields are identified.

![Diagram of Data Cleansing Process](image)

Figure 5.2 Data cleansing process
The purpose of data preprocessing is to extract useful data from raw datasets and then these data should be converted into the format necessary for data cleansing. Due to the irrelevant information and data format, the original raw data cannot be directly used in the data cleansing procedure, hence in data preprocessing phase, raw data need to be cleaned, analyzed and transformed for further step. Consider a sample dataset, where, the attributes presented are name, address, pin code, contact number and email-id. There are total $N$ records such that each record contains fields {“name”, “address”, “pin code”, “contact number”, “E-mail id”}. There are duplicate records, near-duplicate records, missing fields, misspellings in city, illegal values and missing values in pin code, incorrect email-ids and so on. Therefore, before directly going into the cleansing process, it will be very helpful, if undergo some preprocessing steps. The preprocessing consists of two steps such as:

- Remove the records containing three or more empty fields;
- Change all the fields into standard format.

(a) **Remove the records containing two or more empty fields:**
In this step, the records having two or more empty fields are removed since they are not containing the significant information. Consider the data record $R_8$ from table 1 having field values as {Mr. Idris Khan, null, null, 9886060482, null}. In this record, there are 3 null values and there is a chance of presence of more than one Idris Khan in the same data set. Hence, it is a meaningless record and is worth to remove the entire record. Similarly, record $R_4$ has some null values and it is also removed from the data set.

(b) **Change all the fields into standard format:** In this step, the required fields in the records are transformed into standard
format. The name field is divided into first name and last name and, address field is divided into Apartment field, street field and city field. Other fields namely, pin code, contact number and e-mail id is not required any standardization. Thus, after removing unwanted records, obtain \( R = \{ R_1, R_2, ..., R_t \} \), where \( t \) is the total no of records after the preprocessing steps. For example, the name field of \( R_i \) is in the form of (Mr. Rajendra Sharma), which is divided like, (Rajendra) as First name and (Sharma) as Last name after removing the title word (Mr.). Consider the address field, for example, (403 Vandit, Appartment, Bhaikaka Nagar, Thaltej, Ahmedabad). It is converted by dividing it into two separate fields as (403, Vandit, Bhaikaka Nagar, Thaltej) in address field and (Ahmedabad) in city field.

5.1.3 Data Auditing

The data is audited with the use of statistical and database methods to detect anomalies and contradictions. This eventually gives an indication of the characteristics of the anomalies and their locations. Several commercial software packages will let to specify constraints of various kinds (using a grammar that conforms to that of a standard programming language, e.g., JavaScript of Visual Basic) and then generate code that checks the data for violation of these constraints. This process is referred to below in the bullets "workflow specification" and "workflow execution." For users who lack access to high-end cleansing software, Microcomputer database packages such as Microsoft Access or FileMaker Pro will also let to perform such checks, on a constraint-by-constraint basis, interactively with little or no programming required in many cases.
Workflow Specification

The detection and removal of anomalies is performed by a sequence of operations on the data known as the workflow. It is specified after the process of auditing the data and is crucial in achieving the end product of high-quality data. In order to achieve a proper workflow, the causes of the anomalies and errors in the data have to be closely considered.

Workflow Execution

In this stage, the workflow is executed after its specification is complete and its correctness is verified. The implementation of the workflow should be efficient, even on large sets of data, which inevitably poses a trade-off because the execution of a data-cleansing operation can be computationally expensive.

Post-processing and Controlling

After executing the cleansing workflow, the results are inspected to verify correctness. Data that could not be corrected during execution of the workflow is manually corrected, if possible. The result is a new cycle in the data-cleansing process where the data is audited again to allow the specification of an additional workflow to further cleanse the data by automatic processing.

Parsing

Parsing in data cleansing is performed for the detection of syntax errors. A parser decides whether a string of data is acceptable within the allowed data specification. This is similar to the way a parser works with grammars and languages.
Data Transformation

Data transformation allows the mapping of the data from its given format into the format expected by the appropriate application. This includes value conversions or translation functions, as well as normalizing numeric values to conform to minimum and maximum values.

Duplicate Elimination

Duplicate detection requires an algorithm for determining whether data contains duplicate representations of the same entity. Usually, data is sorted by a key that would bring duplicate entries closer together for faster identification.

Statistical Methods

By analyzing the data using the values of mean, standard deviation, range, or clustering algorithms, it is possible for an expert to find values that are unexpected and thus erroneous. Although the correction of such data is difficult since the true value is not known, it can be resolved by setting the values to an average or other statistical value. Statistical methods can also be used to handle missing values which can be replaced by one or more plausible values, which are usually obtained by extensive data augmentation algorithms.

5.1.4 Challenges and Problems

Error correction and loss of information

The most challenging problem within data cleansing remains the correction of values to remove duplicates and invalid entries. In many cases, the available information on such anomalies is limited and insufficient to
determine the necessary transformations or corrections, leaving the deletion of such entries as a primary solution. The deletion of data, though, leads to loss of information; this loss can be particularly costly if there is a large amount of deleted data.

**Maintenance of Cleansed Data**

Data cleansing is an expensive and time-consuming process. So after having performed data cleansing and achieving a data collection free of errors, one would want to avoid the re-cleansing of data in its entirety after some values in data collection change. The process should only be repeated on values that have changed, this means that a cleansing lineage would need to be kept, which would require efficient data collection and management techniques.

**Data Cleansing in Virtually Integrated Environments**

In virtually integrated sources like IBM’s Discovery Link, the cleansing of data has to be performed every time the data is accessed, which considerably decreases the response time and efficiency.

**Data-cleansing Framework**

In many cases, it will not be possible to derive a complete data-cleansing graph to guide the process in advance. This makes data cleansing an iterative process involving significant exploration and interaction, which may require a framework in the form of a collection of methods for error detection and elimination in addition to data auditing. This can be integrated with other data-processing stages like integration and maintenance.
5.2 DUPLICATE RECORD DETECTION AND DELETION

In the broader domain of data cleansing and data quality, detecting and eliminating duplicated data is a significant problem. Therefore, a considerable quantity of time is spent on duplicates identification and removal tasks. In the proposed work, a preprocessed dataset contains ‘n’ number of records $R=\{R_1, R_2, ..., R_n\}$ where, two types of duplicate records namely, duplicate records and near-duplicate records. The removal of duplicate and near-duplicate records is carried out only in the preferred order such as, 
(a) Choosing appropriate fields 
(b) Duplicate records removal and 
(c) Near-duplicate records removal. Records $R_3$ and $R_7$ given in the Table 5.3 are duplicates where, it can be observed that all the fields are exactly matched and record $R_{10}$ is considered as near duplicate record of $R_3$ since the “Address” field differs. The flow diagram for finding the duplicate and near-duplicate records is given below:

![Diagram](image)

**Figure 5.3** Fingerprinting to eliminate duplicate records
(a) **Choosing appropriate fields:** At first, the user is expected to decide and rank the fields that could be combined to faultlessly distinguish one record from another. In the client’s contact data set, the appropriate fields are chosen manually such as, “E-mail Id”, “Pin code”, “First name”, “Address”. For instance, the chosen fields for record \( R_s \) are \{immortal0004@gmail.com 560003 Raghevendra 36, 2ndMain, Gayathri Devi\}.

(b) **Duplicate records removal:** For detecting and removing duplicate records, at first, generates a \( n \)-bit fingerprint by employing Rabin’s Fingerprint algorithm which have been explained already in section 4.1 for a particular record \( R_i \) where, \( i = \{1,2,3,...,n\} \). For example, if apply Rabin’s Algorithm for the record \( R_7 \), “immortal0004 560003 Raghevendra 36 2ndMain Gayathri Devi” is given as an input and it generates \( n \)-bit fingerprint. Now, calculate the frequency of each \( n \)-bit fingerprint generated and observe the records which are having the frequency more than one. With the help of index values, remove the exact duplicate records in such a way that the index values with highest frequency are considered as duplicates. Thus, only the duplicate records is detected and removed effectively.

(c) **Near-duplicate records removal:** In the previous step, it had removed the duplicate records using Rabin’s fingerprinting algorithm. For detecting and removing near-duplicates, make use of Levenshtein distance explained the edit distance between two chosen fields of records is computed. Levenshtein distance is defined for strings of arbitrary length
and obtained by finding the cheapest way to convert one string into another. If string $s$ is “immortal0004 560003 Ragheendra 36, 2ndMain, Gayathri Devi” and string $t$ is “immortal0004 560003 Ragheendra 36, 2ndMain, Gayathri Devi”, then the edit distance obtained in between these two strings is 0. With the help of a threshold value (say $c$), detect the near-duplicate records and remove it from the database. Thus, the dataset after the removal of duplicate records and near-duplicate records, it gets $K = \{K_1, K_2, \ldots, K_n\}$ where $K_i$ is a record, $z = \{1, 2, 3, \ldots, n\}$ and $n$ is the total number of records after duplicate and near duplicates removal.

5.3 CORRECTING MISSPELLING ERRORS

Spelling error detection and correction is a task that identifies misspellings in texts and offer recommendations to fix them. When considering the taken datasets, the misspelling errors might be observed in the text fields namely “City”. Hence, for each record, “city” field is chosen for computing edit distance. For example, consider the “city” fields containing strings, “Bangalore”, “Bengalore” and “Bangalooe”. If $s$ is “Bangalore” and $t$ is “Bangalore”, distance between these two strings, $LD(s,t)$ is zero. So, no transformations are needed because two strings are already identical. If $s$ is “Bangalore” and $t$ is “Bengalore”, then $LD(s,t)$ is 1 and if $s$ is “Bengalore” and $t$ is “Bangalooe”, then $LD(s,t)$ is 2. In similar fashion, the edit distances between all the “city” fields are computed and with the help of a threshold value and frequency, identify the misspelling errors and correct it. For example, “Bangalooe” and “Bengalore” are considered as misspelling error which has smallest frequency and hence, they are replaced by “Bangalore” which has highest frequency.
5.4 FINDING ILLEGAL ERRORS

A set of acceptable values within a certain domain range for a field is considered as legal values. The range of values going beyond a domain range is considered as illegal values. For instance, the pin code must have only 6 digits. If there are more than 6 digits and less than six digits, then it is considered as illegal value error. The illegal value errors of the taken dataset can occur only in “Pin code” and “Contact number” fields. The finding of illegal value error can be obtained using the rules given in the rule base. For example, if the pin code “560008” is typed as “65600” or “5600080”, then it is displayed as illegal value error. Similarly, if the contact number “9986222999” is typed as “99862222” or “99862229990”, then it is considered as illegal value error.

5.5 RABIN’S FINGERPRINTING ALGORITHM

Fingerprints are tiny tags for bigger objects and the non-existence of identical fingerprints for two distinct strings has been guaranteed by sound provable probabilistic. Other checksum algorithms, for example Message Digest 5 (MD5) and Secure Hash Algorithm (SHA) are more expensive to calculate than Rabin Fingerprints. Compared to arbitrary hash function, choosing Rabin fingerprints (which are based on random irreducible polynomials) is advantageous because their probability of collision is well understood. An arbitrarily selected prime $p$ is used to generate “fingerprint”, a long character-string by calculating the residue of that string, perceived as a large integer module $p$ (Micsik & Pajdla 2006).

The general working of Rabin’s Fingerprinting Algorithm (Levenshtein 1966, Bledsoe 1966) by Michael Rabin is as follows: Consider a character string $S$, where, $S = S_{ij} S_{im} \cdots S_{i}$ containing $m$ bits. A $k$-bit Rabin fingerprint for a string $S$ is computed as follows:
\[ S(y) = S_m y^{m-1} + S_{m-1} y^{m-2} + \cdots + S_2 y + S_1 \]  

(5.1)

Then, choose an irreducible polynomial \( P(y) \) of degree \( k \).

\[ P(y) = P_k y^k + P_{k-1} y^{k-1} + \cdots + P_0 \]  

(5.2)

Rabin's fingerprint \( f(S) \) is computed for the string \( S \) is based on the above two equations.

\[ f(S) = S(y) \mod P(y) \]  

(5.3)

The Rabin fingerprinting scheme is a method for implementing fingerprints using polynomials over a finite field. The Rabin fingerprinting algorithm calculates a rolling checksum over data (a file to store in venti). The window of the data to look at is configurable, but typically a few dozen bytes long. The Rabin module will read through a file, and let the window “slide” over the data, recalculating the fingerprint each time when advancing a byte. When the fingerprint assumes a special value, the Rabin module considers the corresponding window position to be a boundary. All data preceding this window position is taken to be a “block” of the file. Calculating a checksum relative to a previous one is not very expensive cpu-wise.

The operations involved are:

Multiplying the previous checksum by a prime (the prime is a parameter of the algorithm). Adding the value of the byte that has just come into the sliding window. Subtracting the value of the byte that has just slid out of the window times the \( n \)th power of the prime (where \( n \) is the width of the sliding window, the 256 possible values are pre-calculated during module initialization). Taking the modulo of the value by the average desired block size (for now, the average desired block size is required to be a power of two).
Check whether the new checksum has the special value that makes it a boundary. This means the algorithm has three parameters: prime, width and modulo. The properties that make this algorithm useful for the purpose is that: 1. they are cheap to calculate; 2. they find the same boundaries, no matter where in the file they occur. Thus, when a byte has been pretended to or inserted into a file, this has no influence on how later blocks are formed, as opposed to the case where all block boundaries are at fixed file offsets. This implementation also allows to specify a minimum and maximum block size. If a block boundary occurs before the minimum block size is encountered, it is ignored. If no block boundary occurs before the maximum block size, the window is treated as a block boundary anyway and a new block emitted. Rabin’s fingerprinting scheme is based on arithmetic modulo an irreducible polynomial with coefficients in $Z_2$.

**Properties of Rabin’s Scheme**

At the hardware level the representation of the string $A$ and the polynomial $A(t)$ with coefficient over $Z_2$ is identical. The basic operations with polynomials have simple implementations: addition is equivalent with bit-wise exclusive or, and multiplication by $t$ is equivalent with shift left one bit.

Fingerprinting is distributive over addition (in $Z_2$): 

$$f(A+B) = f(A) + f(B).$$

Fingerprints can be computed in linear time. More generally, the fingerprint of the concatenation of two strings can be computed via the equality

$$f(\text{concat}(A,B)) = f(\text{concat}(f(A),B)).$$
If it can be given \( f(A) \) and \( f(B) \), and the length \( l \) of \( B \) then

\[
f(\text{concat}(A, B)) = A(t)^l f(t) + B(t) \mod P(t) = f(A)^l f(t) + f(B)
\]

### 5.5.1 Bound of Error

Rabin finds a bound for error in the steps as follows. The number of irreducible polynomial \( P(t) \) of degree \( k \) is

\[
\frac{2^{k-2}}{k} \approx \frac{2^k}{k}
\]

(5.4)

Given a dataset \( S \) containing \( n \) character strings of maximum length \( m \) bits, construct a polynomial

\[
Q(t) = \pi \prod_{\{A, B \in S\}} (A(t) - B(t))
\]

(5.5)

If degree of \( Q(t) = \deg Q \), then

\[
\deg Q = \sum_{\{A, B \in S\}} \deg (A(t) - B(t))
\]

\[
\leq \sum_{\{A, B \in S\}} \max \{\deg A(t), \deg B(t)\}
\]

\[
\leq \sum_{\{A, B \in S\}} m
\]

\[
\leq n^2 m
\]

Maximum number of irreducible factors of degree \( k \) of

\[
Q(t) = \frac{\deg Q}{k} \leq \frac{n^2 m}{k}
\]

(5.6)

For \( f(A) = f(B) \) given \( A \neq B \), one must have \( P(t)(A(t) - B(t)) \) and \( P(t) | Q(t) \).
Probability of making an error = probability of picking a factor of $Q(t)$

The bound of error is estimated as

$$\Pr \{ f(A) = f(B) | A \neq B \} \approx \frac{\deg Q}{k} \cdot \frac{2^{k-2}}{k}$$

(5.7)

$$\approx \frac{\deg Q}{2^k}$$

$$\leq \frac{n^2 m}{2^k}$$

Rabin suggests two ways to lower the probability of error.

The probability of a wrong output will be lowered by increasing the value of $k$. This will require a larger word-length. The probability can also be lowered by using two different irreducible polynomials $P_1(t)$ and $P_2(t)$ of the same degree $k$. This algorithms is then run twice by interleaving steps, one time with $P_1(t)$ and another time with $P_2(t)$. Since the error probabilities are independent, the maximum probability of collision becomes

$$\frac{n^2 m}{2^k}$$

It is begin with a simple version of P assuming the alphabet $x1D49C_EuclidMathOne_10n_000100 = \{0, 1\}$. The fingerprints compared the hashing values e16ue003 and e16ue004 of course, $x \subseteq y$ if and only if $H(x) = H(y[i \ldots i + m])$ for some index $i$ with $0 \leq i \leq n - m$. Of course, comparing the fingerprints $H(y[i \ldots i - 1, i - 1 + m])$ and $H(y[i \ldots i + m])$ may not necessarily be an improvement over comparing $x$ and $y[i \ldots i + m]$ if the calculation of $H(y[i \ldots i + m])$ for different values of $i$ must be made afresh, independent from one another. The Karp-Rabin fingerprint algorithm performs a shifting hash that recursively calculates the hash value $H(y[i \ldots i + m])$, reducing the fingerprint overhead. Rabin’s fingerprinting algorithm is the
prototype of the class. It is fast and easy to implement, allows compounding, and comes with a mathematically precise analysis of the probability of collision. Namely, the probability of two strings r and s yielding the same w-bit fingerprint does not exceed max (|r|, |s|)/2w-1, where |r| denotes the length of r in bits. The algorithm requires the previous choice of a w-bit internal "key", and this guarantee holds as long as the strings r and s are chosen without knowledge of the key. The idea of generating a more flexible and robust fingerprint for binary data was proposed by Rabin. Since then, considerable research has focused on developing ever more sophisticated fingerprinting techniques, but Rabin's basic idea has carried over with relatively small variations. Limiting the discussion to the essential ideas. Interested readers are referred to (Schimke et al 2004) for a detailed survey of hashing and fingerprinting techniques. Rabin's scheme is based on random polynomials and its original purpose was to produce a very simple real-time string matching algorithm and a procedure for securing less against unauthorized changes. A Rabin fingerprint can be viewed as a checksum with low, quantifiable collision probabilities that can be used to efficiently detect identical objects. In the 1990s, there was a renewed interest in Rabin's work in the context of finding similar objects, with an emphasis on text.

Broder et al (2000) applied it to find syntactic similarities in web pages. The basic idea, which is referred to as anchoring, chunking or shingling, is to use a sliding Rabin fingerprint over a fixed-size window that splits data into pieces. A hash value h is computed for every window of size w. The value is divided by a constant c and the remainder is compared with another constant m. If the two values are equal (i.e. = h mod c), then the data in the window is declared as the beginning of a chunk (anchor) and the sliding window is moved one position. This process is continued until the end of the data is reached. For convenience, the value of c is typically a power of two (c = 2k) and m is a fixed number between zero and c = 1. Once the baseline anchoring is determined, it can be used in a number of ways to select
characteristic features. For example, the chunks in between anchors can be chosen as features. Alternatively, the I bytes starting at the anchor positions may be chosen as features, or multiple, nested features may be employed. Note that, while shingling schemes pick a randomized sample of features, they are deterministic and, given the same input, produce the exact same features. Further, they are locally sensitive in that the determination of an anchor point depends only on the previous w bytes of input, where w could be as small as a few bytes. This property can be used to solve the fragility problem in traditional file- and block-based hashing. Consider two versions of the same document. One document can be viewed as being derived from the other by inserting and deleting characters. For example, an HTML page can be converted to plain text by removing all the HTML tags. Clearly, this would modify a number of features, but the chunks of unformatted text would remain intact and produce some of the original features, permitting the two versions of the document to be automatically correlated. For the actual feature comparison, the hash values of the selected features are stored and used as a space-efficient representation of a fingerprint.\textsuperscript{12} The Levenshtein algorithm (also called Edit-Distance) calculates the least number of edit operations that are necessary to modify one string to obtain another string. The most common way of calculating this is by the dynamic programming approach. A matrix is initialized measuring in the (m, n)-cell the Levenshtein distance between the m-character prefix of one with the n-prefix of the other word. The matrix can be filled from the upper left to the lower right corner. Each jump horizontally or vertically corresponds to an insert or a delete, respectively. The cost is normally set to 1 for each of the operations. The diagonal jump can cost either one, if the two characters in the row and column do not match or 0, if they do. Each cell always minimizes the cost locally. This way the number in the lower right corner is the Levenshtein distance between both words. Here is an example that features the comparison of "meilenstein" and "levenshtein".
Table 5.1 Meilenstein and Levenhtein

<table>
<thead>
<tr>
<th>m</th>
<th>e</th>
<th>i</th>
<th>L</th>
<th>6</th>
<th>n</th>
<th>S</th>
<th>t</th>
<th>6</th>
<th>i</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>L</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>c</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>C</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>V</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>E</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>N</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>S</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>H</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>T</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>E</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>I</td>
<td>10</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>N</td>
<td>11</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>

There are two possible paths through the matrix that actually produce the least cost solution. Namely

```
levenshtein   meilenstein
e o o o o o o o o o o o
m e i e i e i e i e i
```

"-" Match, "o" Substitution, "+" Insertion, ";-" Deletion

Though there are sophisticated improvements on the complexity, there is no alternative to calculating the matrix to at least a large extent. Edit distance is a way of comparing two strings with each other. One common use is just to calculate the distance between the two strings, where the distance is the sum of all the replacement, insertions and deletions needed to transform the first string to the other. Given two character strings $S_1$ and $S_2$, the edit distance between them is the minimum number of edit operations required to transform $S_1$ into $S_2$. Most commonly, the edit operations allowed for this purpose are: (i) insert a character into a string; (ii) delete a character from a string and (iii) replace a character of a string by another character; for these operations, edit distance is sometimes known as Levenshtein distance. For
example, the edit distance between cat and dog is 3. In fact, the notion of edit
distance can be generalized to allowing different weights for different kinds
of edit operations, for instance a higher weight may be placed on replacing the
character s by the character p, than on replacing it by the character a (the latter
being closer to s on the keyboard).

Setting weights in this way depending on the likelihood of letters
substituting for each other is very effective in practice. However, the
remainder of the treatment here will focus on the case in which all edit
operations have the same weight. It is well-known how to compute the
(weighted) edit distance between two strings in time $O(|S_1|\cdot|S_2|)$, where $|S_i|
$ denotes the length of a string $S_i$, where the characters in $S_1$ and $S_2$ are given in
array form. The algorithm fills the (integer) entries in a matrix $W$ whose two
dimensions equal the lengths of the two strings whose edit distances is being
computed; the $(i,j)$ entry of the matrix will hold (after the algorithm is
executed) the edit distance between the strings consisting of the first $i$
characters of and the first $j$ characters of $S_2$, where the three quantities whose
minimum is taken correspond to substituting a character in $S_1$, inserting a
character in $S_1$ and inserting a character in $S_2$. Edit distance algorithm is used
below

$$
\text{EditDistance}(s_1, s_2)
$$

```python
int m[i,j]=0
for i ← 1 to |s1|
do m[i,0]=i
for j ← 1 to |s2|
do m[0,j]=j
```
for $i \leftarrow 1$ to $|s_1|

do for $j \leftarrow 1$ to $|s_2|

\quad do m[i,j]=\min\{m[i-1,j-1]+$ \text{if}(s_1[i]=s_2[j]) \text{then} 0$ \text{else} 1,$
\quad \quad \quad m[i-1,j]+1,$
\quad \quad \quad m[i,j-1]+1\}$

return $m[|s_1|,|s_2|]$

Table 5.2 Generation of fingerprints

<table>
<thead>
<tr>
<th>Record</th>
<th>Fingerprints</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>010011111000110101010001101001</td>
</tr>
<tr>
<td>5</td>
<td>1101100001000101100111110101</td>
</tr>
<tr>
<td>7</td>
<td>1101100001000101100111110101</td>
</tr>
<tr>
<td>10</td>
<td>1101100001000101100111110100</td>
</tr>
</tbody>
</table>

Table 5.3 Sample records containing exact and near-duplicates

<table>
<thead>
<tr>
<th>R. No</th>
<th>First Name</th>
<th>Last Name</th>
<th>Address</th>
<th>City</th>
<th>Pin code</th>
<th>Contact No</th>
<th>E-mail Id</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Harihan</td>
<td></td>
<td>101, Shriram Complex, Nr Silicon Tower</td>
<td>Ahmedabad</td>
<td>38000945</td>
<td>9976124916 9820217300</td>
<td><a href="mailto:harihan@gmail.com">harihan@gmail.com</a></td>
</tr>
<tr>
<td>5</td>
<td>Ragheendra</td>
<td>M</td>
<td>56, 2nd Main, Gayaun Devi</td>
<td>Bangalore</td>
<td>560003</td>
<td>98925531242</td>
<td><a href="mailto:pranav@gmail.com">pranav@gmail.com</a></td>
</tr>
<tr>
<td>7</td>
<td>Ragheendra</td>
<td>M</td>
<td>56, 2nd Main, Gayaun Devi</td>
<td>Bangalore</td>
<td>560003</td>
<td>98925531242</td>
<td><a href="mailto:pranav@gmail.com">pranav@gmail.com</a></td>
</tr>
<tr>
<td>10</td>
<td>Harihan</td>
<td></td>
<td>102, Shriram Complex, Nr Silicon Tower</td>
<td>Ahmedabad</td>
<td>380009</td>
<td>9976124916 9820217300</td>
<td><a href="mailto:harihan@gmail.com">harihan@gmail.com</a></td>
</tr>
</tbody>
</table>

Table 5.3 shows the sample records containing exact-duplicates and near-duplicates

Dynamic programming algorithm for computing the edit distance between strings $s_1$ and $s_2$. The typical cell $(i,j)$ has four entries formatted as a
The lower right entry in each cell is the \( \min \) of the other three, corresponding to the main dynamic programming. The other three entries are the three entries \( m[i-1,j-1]+0 \) or \( 1 \) depending on whether \( s_1[i]=s_2[j] \), \( m[i-1,j-1]+1 \) and \( m[i,j-1]+1 \). The cells with numbers in italics depict the path by which determine the Levenshtein distance.

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>a</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>t</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>s</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

![Figure 5.4 Example Levenshtein distance computation](image)

The spelling correction problem however demands more than computing edit distance: given a set \( S \) of strings (corresponding to terms in the vocabulary) and a query string \( q \), seek the string(s) in \( \mathcal{V} \) of least edit distance from \( q \). It may view this as a decoding problem, in which the codewords (the strings in \( \mathcal{V} \)) are prescribed in advance. The obvious way of doing this is to compute the edit distance from \( q \) to each string in \( \mathcal{V} \), before selecting the string(s) of minimum edit distance. This exhaustive search is inordinately expensive. Accordingly, a number of heuristics are used in practice to efficiently retrieve vocabulary terms likely to have low edit distance to the query term(s).

The simplest such heuristic is to restrict the search to dictionary terms beginning with the same letter as the query string; the hope would be
that spelling errors do not occur in the first character of the query. A more sophisticated variant of this heuristic is to use a version of the permuteerm index, in which omitting the end-of-word symbol $. Consider the set of all rotations of the query string $q$. For each rotation $r$ from this set, traverse the B-tree into the permuteerm index, thereby retrieving all dictionary terms that have a rotation beginning with $r$. For instance, if $q$ is mase and consider the rotation $r = sema$, it would be retrieve dictionary terms such as semantic and semaphore that do not have a small edit distance to $q$. Unfortunately, it would miss more pertinent dictionary terms such as mare and mane. To address this, refine this rotation scheme: for each rotation, omit a suffix of $l$ characters before performing the B-tree traversal. This ensures that each term in the set $R$ of terms retrieved from the dictionary includes a "long" substring in common with $q$. The value of $l$ could depend on the length of $q$. Alternatively, it may set it to a fixed constant such as 2.

The implementations of the Levenshtein algorithm on this page are illustrative only. Applications will, in most cases, use implementations which use heap allocations sparingly, in particular when large lists of words are compared to each other.

The following remarks indicate some of the variations on this and related topics:

- Most implementations use one- or two-dimensional arrays to store the distances of prefixes of the words compared. In most applications the size of these structures is previously known. This is the case, when, for instance the distance is relevant only if it is below a certain maximally allowed distance (this happens when words are selected from a dictionary to
approximately match a given word). In this case the arrays can be pre allocated and reused over the various runs of the algorithm over successive words.

- Using a maximum allowed distance puts an upper bound on the search time. The search can be stopped as soon as the minimum Levenshtein distance between prefixes of the strings exceeds the maximum allowed distance.

- Deletion, insertion, and replacement of characters can be assigned different weights. The usual choice is to set all three weights to 1. Different values for these weights allows for more flexible search strategies in lists of words.

Edit distance, also known as Levenshtein distance or evolutionary distance is a concept from information retrieval and it describes the number of edits (insertions, deletions and substitutions) that have to be made in order to change one string to another. It is the most common measure to expose the dissimilarity between two strings (Levenshtein 1966; Navarro & Raffinot 2002). The edit distance \(ed(x, y)\) between strings \(x=x_1 \ldots x_m\) and \(y=y_1 \ldots y_n\), where \(x, y\)

\[\in \Sigma^*\] is the minimum cost of a sequence of editing steps required to convert \(x\) into \(y\).

The alphabet \(\Sigma\) of possible characters \(ch\) gives \(\Sigma^*\), the set of all possible sequences of \(ch \in \Sigma\). Edit distance can be calculated using dynamic programming. Dynamic programming is a method of solving a large problem by regarding the problem as the sum of the solution to its recursively solved sub problems. Dynamic programming is different to recursion however. In order to avoid recalculating the solutions to sub problems, dynamic
programming makes use of a technique called memoisation, whereby the
solutions to sub problems are stored once calculated, to save recalculation.

5.5.2 Edit Distance Algorithms

There are a number of metrics available to achieve the string
matching tasks but the basic metrics are based on ED metrics. Various ED
metrics have been developed so far to decrease the penalty for the most
possible transcription errors (Jogan & Leonardis 2003, Lourakis & Argyros
2004). The main problem is how to select or combine multiple orthographic
measures (Lourakis & Argyros 2004) in order to achieve desired results. The
basic EDA’s are based on dynamic programming including Smith-Waterman,
Levenshtein Distance and Needleman Wunsch (Micusik & Pajdla 2006).
These dynamic programming algorithms needs \( O(m \times n) \) operations to
calculate the edit distance between two strings, where ‘m’ and ‘n’ are the
lengths of string1 and string2, respectively. Dynamic programming generates
the \((m + 1) \times (n + 1)\) matrix and compute all values of \(D(i, j)\) by using a
recursive function and stores the result in a table, where ‘i’ and ‘j’ represents
all strings from ‘1 to m/n’.

The LEDA counts the minimum number of edit operations required
to transform one string to another (Rousselieu 1984, Micusik & Pajdla
2006). It is also referred as basic Levenshtein (BLev) EDA. The LEDA
allows three basic edit operations as given below:

1) Insert: \(D(i-1, j) + 1\)

2) Delete: \(D(i, j-1) + 1\)

3) Substitute: \(D(i-1, j-1) + \text{Cost}\)

If \(ai = bj\) then Cost=0 and if \(ai \neq bj\) then Cost=1
Introducing an additional new edit operation, that is, ‘exchange of vowels’ (a, e, i, o, u). This edit operation is proposed to find the most commonly occurring orthographic and typographical errors especially in person names. The ‘exchange of vowels’ edit operation is introduced to account for the most commonly occurring spelling mistakes of vowels due to the converting names from one language to another.

The Damerau-Levenshtein distance metric arose from research by Damerau and Levenshtein on spelling errors (Damerau 1964, Levenshtein 1966). The Damerau-Levenshtein distance metric is a function, from finite strings drawn from an alphabet, to the integers. It is a distance metric in the sense. That given strings s1, s2, s3, the following conditions apply.

- Non-negativity: \( d(s_1, s_2) \geq 0 \).
- Non-degeneracy: \( d(s_1, s_2) = 0 \) if and only if \( s_1 = s_2 \).
- Symmetry: \( d(s_1, s_2) = d(s_2, s_1) \).
- Triangle Inequality: \( d(s_1, s_2) + d(s_2, s_3) \geq d(s_1, s_3) \).