МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Чорноморський національний університет
імені Петра Могили
Факультет комп'ютерних наук
Кафедра інтелектуальних інформаційних систем

ДОПУЩЕНО ДО ЗАХИСТУ
Завідувач кафедри інтелектуальних інформаційних систем, д-р техн. наук, проф.
_____Ю. П. Кондратенко
«____»_____2024 р.

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

ІНТЕЛЕКТУАЛЬНА СИСТЕМА ОБРОБКИ НЕСТРУКТУРОВАНИХ ТЕКСТОВИХ ДАНИХ НА ОСНОВІ ТЕХНОЛОГІЇ JSON

Спеціальність 122 «Комп'ютерні науки»

122 – КРМ– 601.21810316

Виконав студент 6-го курсу, групи 601
_____ Є. А. Кучеренко
«20» лютого 2024 р.

Керівник: канд. фіз.-мат. наук, доцент
_____І. В. Кулаковська
«20» червня 2024 р.

Миколаїв – 2024

Чорноморський національний університет ім. Петра Могили
Факультет комп'ютерних наук
Кафедра інтелектуальних інформаційних систем

Освітньо-кваліфікаційний рівень **магістр**
Галузь знань **12 «Інформаційні технології»**
(шифр і назва)
Спеціальність **122 «Комп'ютерні науки»**
(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри інтелектуальних інформаційних систем, д-р техн. наук, проф.
_____ Ю. П. Кондратенко
«____» _____20___р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи

Кучеренко Єгору Андрійовичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи «Інтелектуальна система обробки неструктурованих текстових даних на основі технології JSON».
Керівник роботи Кулаковська Інесса Василівна, канд. фіз.-мат. наук, доцент.
Затв. наказом Ректора ЧНУ ім. Петра Могили від «___» _____ 20___ р. № ____
2. Строк представлення кваліфікаційної роботи студентом «___» _____ 20__ р.
3. Вхідні (початкові) дані до роботи: слабоструктуровані дані у форматі JSON.
4. Перелік питань, що підлягають розробці (зміст пояснювальної записки):

– аналіз методів та систем для автоматизованої обробки великих масивів даних;

– принципи автоматичного розбиття та обробки текстових файлів, розмір яких перевищує доступний об'єм ОЗП;

– визначення та обгрунтування основних принципів та методик роботи з слабоструктурованими текстовими файлами у форматі CSV (Comma Separated Value);

– демонстрація результатів валідації та обробки текстових даних, які відносяться до різних галузей.

5. Перелік графічного матеріалу: презентація.

6. Завдання до спеціальної частини: «Аналіз вимог до умов праці, заходи з техніки безпеки та охорона праці під час роботи за комп'ютером».

7. Консультанти розділів роботи.

Розділ	Прізвище, ініціали та посада консультанта	Підпис
Спеціальна частина з охорони праці	**д-р біол. наук, професор Григор'єва Л. І.**	
Методична частина	**канд. фіз.-мат. наук, доцент Кулаковська І. В.**	

Керівник роботи канд. фіз.-мат. наук, доцент Кулаковська І. В._____
(наук. ступінь, вчене звання, прізвище та ініціали)

_____
(підпис)

Завдання прийнято до виконання Кучеренко Є. А._____
(прізвище та ініціали)

_____
(підпис)

Дата видачі завдання «_31_» ____жовтня_____ 2023___ р.

КАЛЕНДАРНИЙ ПЛАН
виконання кваліфікаційної роботи

Тема: _Інтелектуальна система обробки неструктурованих текстових даних на основі технології JSON

№	Найменування роботи	Початок	Закінчення	Примітки
1	Визначення керівника і теми КРМ. Подання заяви на затвердження теми КРМ	01.09.2023	10.10.2023	Виконано
2	Отримання завдання на виконання КРМ	11.10.2023	01.11.2023	Виконано
3	Складання календарного плану на період виконання КРМ	02.11.2023	10.11.2023	Виконано
4	Огляд літератури за темою дослідження	11.11.2023	26.11.2023	Виконано
5	Проходження передатестаційної практики, збір та аналіз матеріалів до КРМ	27.11.2023	23.12.2023	Виконано
6	Аналіз предметної області та розробка технічного завдання. Моделювання результатів	25.12.2023	12.01.2024	Виконано
7	Опис фахової частини КРМ	13.01.2024	25.01.2024	Виконано
8	Розробка спеціальної частини з охорони праці та методичної частини	26.01.2024	02.02.2024	Виконано
9	Перший попередній захист КРМ на засіданні комісії кафедри	29.01.2024	29.01.2024	Виконано
10	Корегування роботи за результатами попереднього захисту	30.01.2024	05.02.2024	Виконано
11	Доробка та остаточне оформлення КРМ	06.02.2024	11.02.2024	Виконано
12	Другий попередній захист КРМ на засіданні комісії кафедри	12.02.2024	12.02.2024	Виконано
13	Подання КРМ, її електронної копії та інших документів (відгуку, рецензії) до захисту	19.02.2024	20.02.2024	Виконано
14	Захист КРМ перед екзаменаційною комісією (ЕК)	26.02.2024	27.02.2024	Виконано

Розробив студент _____Кучеренко Є. А _____ _____
(прізвище та ініціали) (*підпис*)

Керівник роботи канд. фіз.-мат. наук, доцент Кулаковська І. В.___

_____
(наук. ступінь, вчене звання, прізвище та ініціали) *(підпис)*

«09» __листопада_____ 2023 р.

ВІДГУК

на кваліфікаційну роботу магістра групи 601

Чорноморського національного університету імені Петра Могили

Кучеренко Єгора Андрійовича

«Інтелектуальна система обробки неструктурованих текстових даних на основі технології JSON»

Кваліфікаційна робота магістра (КРМ), яку виконав студент Кучеренко Є. А., присвячена актуальній проблемі автоматизації обробки великих масивів даних з декількох джерел

В процесі виконання роботи проведено аналіз методів та систем для автоматизованої обробки великих масивів даних. Обґрунтовано використання зазначених технологій та наведено приклади їх взаємодії задля покращення результатів обробки великих масивів слабоструктурованої інформації. Розглянуто процес автоматизації систем для структуризації даних з декількох джерел. Розроблено алгоритм обробки великих масивів даних, розмір яких перевищує доступний об’єм ОЗП та надано визначення та обґрунтування основних принципів та методик роботи з слабоструктурованими текстовими файлами у форматі CSV (Comma Separated Value). Застосунок написаний за допомогою мови програмування Python. Розроблений Застосунок працює за допомогою встановлення системи віртуальних оточень virtualenv на операційних системах Microsoft Windows та Linux.

Треба відмітити високі ділові якості студента Кучеренка Є. А., його підготовленість до самостійної роботи, високу працездатність та організованість. При узагальнені результатів кваліфікаційної роботи магістра її автор проявив здатність застосовувати теоретичні знання для розроблення, оформлення та представлення навчальних і наукових робіт українською та англійською мовами.

Враховуючи усе вищесказане вважаю за можливе допустити роботу Кучеренка Є. А. до захисту та присвоїти освітню кваліфікацію «Магістр з комп’ютерних наук (Магістр з комп’ютерних наук)» в галузі знань 12 «Інформаційні технології» за спеціальності 122 «Комп’ютерні науки» (122 «Комп’ютерні науки»).

Керівник роботи, канд. фіз.-мат. наук, доцент І. В. Кулаковська

АНОТАЦІЯ
до кваліфікаційної роботи магістра
студента групи 601 ЧНУ ім. Петра Могили

Кучеренко Єгора Андрійовича

на тему: **«ІНТЕЛЕКТУАЛЬНА СИСТЕМА ОБРОБКИ НЕСТРУКТУРОВАНИХ ТЕКСТОВИХ ДАНИХ НА ОСНОВІ ТЕХНОЛОГІЇ JSON»**

Актуальність даного дослідження полягає у необхідності автоматизації аналізу витоків персональних даних. Практична значимість розробленої системи полягає у вдосконаленні методів збору та обробки інформації з метою її подальшої валідації, очистки та накопичення за наступними категоріями: Паспортні дані, географічні адреси та гео-координати, валідація та автоматизоване доповнення номеру мобільного телефону до міжнародного формату, Обробка автомобільних номерів (у сучасному та застарілому форматі), VIN-коду двигуна та марки автомобіля, валідація url-адрес соціальних мереж, Обробка персональних даних (ПІБ, ДН).

Об'єктом дослідження є процес автоматизації систем для структуризації даних з декількох джерел.

Предметом дослідження є методи та алгоритми реалізації цілісної системи для виконання автоматизованої та паралельної обробки, валідації та структуризації даних.

Метою є розробка системи для підвищення ефективності автоматизації обробки великих даних.

В результаті виконання роботи було досліджено два мета-евристичних методи оптимізації (метод мурашиної і штучної бджолиної колоній), проаналізовано вплив їх внутрішніх параметрів на роботу алгоритмів, визначені основні їх переваги та недоліки, а також розроблено програмне забезпечення, в якому реалізовані відповідні методи.

Дана робота складається з трьох розділів. У першому розділі представлено огляд способів зчитування вхідних даних, описано особливості розбиття та першочергової підготовки даних та основні кроки для роботи з файлами або чергою файлів, загальний об’єм дискового простору для яких – перевищує доступний ресурс ПЗУ, куди і будуть завантажені дані під час обробки.

Другий розділ включає в себе опис технологій та процесів «маппінгу» даних та взаємозалежність між процедурою маппінгу та функціями обробки даних за допомогою спеціалізованих функцій-юнітів, кожна з яких може використовуватись у процесі очищення одного чи декількох стовпців даних – незалежно одна від одної або у комбінованому форматі, колі кожна наступна функція та маніпуляція з даними – базується на результаті виконання попередньої операції, що в свою чергу впливає на їх загальний порядок та швидкість обробки даних.

У третьому розділі наведено детальний опис внутрішніх механізмів розробленої автоматизованої системи та способів їх взаємодії і функціонування, а саме наведено взаємозв’язки між функціями сирцевого коду та їх детальних опис, приклад створення та параметризації конфігураційного файлу, на базі якого й буде працювати уся система. Окремо наведено детальний опис деяких специфічних та спеціалізованих функцій для обробки даних та деталі їх алгоритмічної реалізації.

У спеціальній частині, присвяченій охороні праці, який складається з двох підрозділів у яких відповідно наведено основні засоби та заходи безпеки для працівників у офісних приміщення відповідно до ДСТУ та питання що стосуються цивільного захисту.

Загальний обсяг роботи – 134 сторінки. Кваліфікаційна робота магітсра не містить додатків, 8 рисунків, не містить таблиць і посилання на 9 літературних джерел.

Ключові слова: валідація, інтелектуальна система, неструктуровані дані, JSON, CSV, краулінг, ETL, ELT, автоматизована система.

ABSTRACT

to the master's qualification work by the student of the group 601 of Petro Mohyla Black Sea National University

Kucherenko Yehor

“INTELLIGENT SYSTEM FOR PROCESSING UNSTRUCTURED TEXT DATA BASED ON JSON TECHNOLOGY”

The relevance of this study lies in the need to automate the analysis of personal data leaks. The practical significance of the developed system is to improve the methods of collecting and processing information for the purpose of its further validation, cleaning and accumulation in the following categories: Passport data, geographical addresses and geo-coordinates, validation and automated completion of a mobile phone number to the international format, Processing of car license plates (in modern and outdated format), VIN-code of the engine and car brand, validation of social networking url-addresses, Processing of personal data (name, date).

The object of research is the process of automating systems for structuring data from multiple sources.

The subject of the study is methods and algorithms for implementing an integrated system for automated and parallel processing, validation and structuring of data.

The goal is to develop a system to increase the efficiency of automating big data processing.

As a result of the work, two meta-heuristic optimization methods (the ant and artificial bee colony methods) were investigated, the influence of their internal parameters on the operation of the algorithms was analyzed, their main advantages and disadvantages were identified, and software was developed in which the corresponding methods were implemented.

This paper consists of three sections. The first section provides an overview of input data reading methods, describes the peculiarities of data partitioning and priority preparation, and the main steps for working with files or a queue of files whose total disk

space exceeds the available ROM resource where the data will be loaded during processing.

The second section includes a description of the technologies and processes of data mapping and the interdependence between the mapping procedure and data processing functions using specialized unit functions, each of which can be used in the process of cleaning one or more data columns - independently of each other or in a combined format, where each subsequent function and data manipulation is based on the result of the previous operation, which in turn affects their overall order and speed of data processing.

The third section provides a detailed description of the internal mechanisms of the developed automated system and the ways in which they interact and function, namely, the interrelationships between the functions of the raw code and their detailed descriptions, an example of creating and parameterizing a configuration file on the basis of which the entire system will operate. A detailed description of some specific and specialized functions for data processing and details of their algorithmic implementation are provided separately.

The special part devoted to labor protection consists of two subsections, which respectively describe the basic means and safety measures for employees in office premises in accordance with DSTU and issues related to civil protection.

The total volume of the work is 134 pages. The master's thesis does not contain any appendices, 8 figures, no tables and references to 9 literary sources.

Keywords: validation, intelligent system, unstructured data, JSON, CSV, crowding, ETL, ELT, automated system.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

ВДТ – візуальний дисплейний термінал

ЕОМ – електронна обчислювальна машина

КПО – коефіцієнт природної освітленості

ОС – операційна система

ПЕОМ – персональні електронні обчислювальні машини

ПЗ – програмне забезпечення

ПК – персональний комп’ютер

ССБП – система стандартів безпеки праці

ЦП – центральний процесор

ФС – файлова система

RAM – Random Access Memory

HDD – Hard Disk Drive

ETL – Extract Transform Load

ELT – Extract Load Transform

ISDN – Integrated Services Digital Network

VIN – Vehicle identification number

TCP – Transmission Common Protocol

IP – Internet Protocol

ВСТУП

ETL і ELT системи - окремий клас підсистем для опрацювання великого масиву даних, який розшифровується як Extract-Transform-Load і Extract-Transform-Load. В абревіатурах та їхніх розшифровках уже містяться основи підходу до роботи з даними, які включають до свого складу три основні пункти.

1. Extract.
2. Transform.
3. Load.

Для ETL і ELT систем – пункти 2 і 3 - тобто «Transform» і «Load» - поміняні місцями, що в деяких випадках може поліпшити або прискорити процес оброблення даних або істотно вплинути на результат залежно від роду першочергових даних, які підлягають аналізу та обробленню.

Кожен з етапів незалежно від їхнього розташування (окрім етапу Extract – який за визначенням має йти першим) - виконує певні функції.

Так, першим етапом як ETL, так і ELT систем є процедура «Extract» – тобто «витяг» даних з будь-якого типу носія, сховища або зовнішнього першоджерела. Як приклади корисної роботи цього етапу є – парсинг – як приклад роботи з «зовнішніми» по відношенню до системи даними, і по відношенню до яких - термін «Extract» – можна вважати роботою допоміжної програми-парсера, що збирає першочергові дані для роботи системи. Іншим прикладом як можна інтерпретувати цей етап - є робота з «внутрішніми» по відношенню до системи - даними - тобто, такими даними, які зберігаються у відносно незмінному фіксованому форматі - тобто не залежать від доступності інших сервісів, наприклад веб-сайту, для якого проводиться процедура парсингу, що робить ці дані більш надійними для подальшої обробки. У такому разі термін «Extract» зазвичай інтерпретують як зчитування даних із зовнішнього носія (наприклад, жорсткого диска) - до оперативної пам’яті пристрою, або зчитування даних із будь-якого виду БД

(реляційних, нереліаційних, векторних БД, озера даних) (Data Lake) - тощо) за допомогою відповідних СУБД.

Актуальність. З початком повномасштабної війни росії проти України – в активну фазу перейшла також і інформаційна війна, яку наразі можна спостерігати – через щоденні витоки інформації, яка публікується на анонімних файлообмінниках та популярних месенджерах – наприклад у Telegram.

Витоки даних включають в себе не тільки пошкодження інфраструткури при цільових атаках, а також загрожують деанонімізацію багатьох українських користувачів – в залежності від чутливості витоку даних та його масштабу. Також, окремим напрямкам розвитку є використання технологій парсингу веб-версій сучасних застосунків та соціальних мереж – наприклад WhatsApp та Telegram.

Метою роботи є розробка системи для підвищення ефективності автоматизації обробки великих даних.

Об’єктом дослідження є процес автоматизації систем для структуризації даних з декількох джерел.

Предметом дослідження є методи та алгоритми реалізації цілісної системи для виконання автоматизованої та паралельної обробки, валідації та структуризації даних.

Як уже було сказано, наступним етапом після вилучення певної частини даних у пам’ять пристрою можуть бути як етапи трансформації даних, так і їхнє завантаження. Більш простою та інтуїтивно зрозумілою є ETL-система – де другим етапом слугує етап трансформації даних. Етап трансформації даних охоплює будь-яку закінчену і детерміновану низку операцій над даними, що призводить до їх фільтрації та/або зміни. Вид, кількість і тип таких операцій над даними - залежить від їхньої першочергової природи та типу - чи є дані, що підлягають трансформації, числовими чи такими, що належать до типу термінового типу, рівню та ступеню їхньої різниці між початковим станом і кінцевим бажаним результатом. Прикладом найбільш елементарної трансформації може слугувати зміна регістру символів для

рядка - наприклад, переведення всіх перших літер кожного слова у верхній регістр - для імен та прізвищ. Слід зазначити, що для певного набору вхідних даних кількість операції трансформації може дорівнювати нулю, якщо ці дані вже відповідають певним показником і бажаному результату. Також важливим зауваженням є те, що операції трансформації над даними - є рефлексивними - тобто сумарна робота з даними може дорівнювати нулю наприкінці роботи, якщо операції над даними «скасовують» одна одну, або іншими словами - виконують протилежні дії над даними - щодо іншої операції.

При розгляді прикладів щодо ETL-систем – останнім пунктом є пункт – «Load» – тобто завантаження (або інакше «вивантаження») вже змінених даних у зовнішнє сховище – найчастіше СУБД, тип якого може не збігатися з типом сховища, з якого дані було вилучено. Для ELT-систем основною різницею є те, що трансформація даних відбувається вже після їхнього завантаження до сховища, що робить такі системи більш стійкими до помилок, адже операції трансформації можна відмінити, повторити або змінити - виконавши їх в іншому порядку, враховуючи їхню можливу нетранзитивність. - просто заново прочитавши їх із зовнішнього сховища, яке вважається безумовно надійним на відміну від зовнішнього сховища, з якого спочатку зчитуються дані - на етапі «Extract», і яке може бути вже недоступне при повторенні трансформацій в інший період часу.

У розробленій автоматизованій системі очищення неструктурованих текстових даних з використанням конфігурованих файлів формату json запропоновано використати створену ELT систему, яка використовує та оперує поняттям «юніт» («unit») - одиничною незалежною одиницею, що здатна опрацьовувати певну кількість неструктурованої інформації, використовуючи перелік правил трансформації, які «прив’язані» до юніта та впливають на вхідні дані.

За такого гранульованого підходу стає можливим виконувати такі маніпуляції з даними як:

– призначення певному юніту - індекси стовпців файлу формату CSV (Comma Separated Value), що підлягає обробці;

– використовувати техніку «чорних» і «білих» списків для фільтрації вхідних значень для кожного юніта на етапі тонізації даних;

– легко розширювати операції трансформації - просто додаючи виклик нових функцій до постійно закріпленого за певним юнітом списку функції трансформації;

– розробити систему "антипатернів", що дає змогу юніту викликати функції інших юнітів, для додаткової перевірки, чи не належать дані та їхнє токінізоване подання - до інших юнітів, і чи не варто в такому разі застосувати до них інші правила трансформації;

– використовувати один юніт «під виглядом іншого» – коли юніт перебирає на себе список функцій трансформацій і всі інші налаштування - іншого юніта.

Описаний підхід має багато переваг, і наведений список не є повним.

Для виконання поставленої мети використовуються такі технології та методи.

Технології:

– мова програмування Python;

– додаткові бібліотеки мови програмування Python - для етапу вилучення даних:

– Selenium – бібліотека для автоматизації імітування взаємодії з графічним користувацьким інтерфейсом (GUI) та їхніми елементами;

– BeautifulSoup4 – бібліотека для парсингу вмісту HTML і XML-файлів;

– Requests - бібліотека для виконання HTTP- і HTTPS-запитів.

Додаткові бібліотеки мови програмування Python - для етапу трансформації даних:

– Json - вбудована в мову програмування Python – стандартна однойменна з форматом файлу – бібліотека;

– XML - вбудована в мову програмування Python – стандартна однойменна з форматом файлу – бібліотека;

– бібліотека filesplit для автоматичної «нарізки» великих за обсягом пам’яті текстових файлів;

додаткові бібліотеки мови програмування Python – для етапу вивантаження оброблених даних:

– бібліотека csv – вбудована в мову програмування Python - стандартна однойменна з форматом файлу – бібліотека;

– бібліотека hashlib – для виконання операції взяття хешу (MD5) від рядка.

Методи:

– ООП – підходи Об’єктно Орієнтованого Програмування;

– динамічний аналіз вільної доступної пам’яті ОЗП (RAM);

– засоби та алгоритми парсингу та скрапінгу даних - алгоритми збирання даних із зовнішніх джерел - таких як веб-сайти. Алгоритми парсингу для вилучення інформації з інших заздалегідь структурованих або слабо структурованих файлів.

Таким чином, розроблена ETL-система підвищує гнучкість роботи з даними, водночас зменшуючи зовнішні втручання користувача завдяки підходу «налаштував і запустив», що дає змогу один раз створити та декларативно описати об’єкт юніта і потім неодноразово використовувати його.

Етапи роботи ETL системи:

– завантаження сирих даних;

– читання даних;

– читання повністю за одну транзакцію;

– читання по частинах;

– розбиття даних на окремі рядки;

– розбиття кожного окремого рядка – на стовпці.

Кожен отриманий стовпець – має на увазі «внутрішнє» розбиття за іншим роздільником:

– так;

– ні.

Для кожного рядка:

– початковий маппінг отриманих стовпців до "юнітів" ETL-системи;

– проведення трансляції сирих даних. Виклик усіх заданих функцій etl-юніта на вхідних сирих даних;

– маппінг очищених стовпців у потрібному порядку – наприклад, об’єднання кількох стовпців в один.

«Схлопування» рядків за заданою ознакою (стовпчиком) – задається стовпець, який не враховується як унікальна ознака, і якщо рядки без урахування даних у цьому стовпчику/стовпчиках однакові, вони вважаються дублікатами.

1 АНАЛІЗ ТА ОГЛЯД МЕТОДІВ ОБРОБКИ ВЕЛИКИХ ДАНИХ

Будь-яку ETL-систему можна уявити як функцію «чорний ящик» – для оброблення файлу, на вхід якої подають файл, його дані піддаються маніпуляціям, а на виході - отримуємо другий файл із даними в потрібному форматі. Таким чином, мінімальною одиницею для роботи ETL-системи – можна вважати файл. Файлом може в цьому разі вважатися будь-який потік даних, у якого є джерело – чи то файл у файловій системі ОС, чи то частина потоку даних, що надходить у реальному часі, до того ж, для потоку файлом вважатиметься та його частина, об’єм якої система може обробити без затримок.

Розглянемо основні характеристики вхідних і вихідних файлів для ETL-системи:

– ім’я файлу (вхідного і вихідного) – в даному випадку, ім’ям файлу може бути як шлях до файлу в ієрархічній ФС, так, і, наприклад, адреса сервера, звідки йде отримання даних по IP / TCP – або будь-якому іншому протоколу;

ім’я файлу може мати такі обмеження:

– список заборонених символів (наприклад спецсимволи);

– якщо ім’я файлу представлено повним шляхом до файлу, а не відносним - можлива відсутність файлу за вказаним шляхом (або, наприклад, сервер, на якому зберігаються вхідні дані, недоступний).

– кодування текстових даних - важливо розуміти, що кодування вхідного і вихідного текстових файлів - не зобов’язане збігатися;

– формат читання даних. До цього параметра можна віднести такі характеристики (які так само не зобов’язані збігатися для вхідного і вихідного файлів):

– тип зазначеного вхідного шляху - чи є зазначений шлях - шляхом до файлу або ж директорії (безлічі файлів). У разі, якщо вказаний шлях - є директорією - необхідно циклічно обробити кожен файл у директорії.

По-суті, роботу з файлами в директорії можна розділити на два етапи:

– отримання списку всіх файлів у заданій директорії;

– здійснення фільтрації файлів у списку - за заданими критеріями, наприклад, за розширенням файлу, тому що не можна бути впевненим, що в директорії не буде "зайвих" або невідповідних файлів, або вони не з’являться в майбутньому.

Обсяг даних, які будуть прочитані за одну ітерацію - це може бути як весь файл цілком, так і певна кількість даних. Цей параметр в основному залежить від обсягу доступної оперативної пам’яті. Вище спеціально використано формулювання *"...так і певну кількість даних"* – тому що під час роботи з текстовими даними важливо розуміти, що об’єм файлу в байтах - може не дорівнювати точній заданій кількості рядків. Таким чином, якщо читання файлу проводиться по-частинах, то межу об’єму файлу - необхідно перевіряти в байтах - відносно об’єму доступної оперативної пам’яті або її частини (наприклад, 1/2, 1/4 і т. п.), і якщо цей об’єм перевищено - необхідно здійснити розбивку файлу, але за кількістю рядків, а не за об’ємом байт, що міститимуться в частинах файлу, які будуть отримані внаслідок розбивки. Під час роботи з файлом – бажано множити його об’єм на два, оскільки необхідно враховувати зберігання в пам’яті не тільки вихідних даних, а й очищених, якщо їх, звісно, не буде записано в проміжні файли, які потім, зі свого боку, будуть склеєно. Множення об’єму вхідних даних на два - гарантує максимальну верхню межу, яку не буде перевищено, - тому що сумарний об’єм вихідних і очищених даних не може перевищувати подвійний об’єм вхідних даних (тільки за умови, що операції очищення даних не мають на увазі додавання нових значень, - наприклад, префікса «Робота:» для стовпця, в якому зазначена посада):

– символ роздільника рядків;

– символ роздільника стовпців у рядку;

– символ роздільника значень усередині одного стовпця (може використовуватися для рекурсивного очищення в межах одного стовпця);

– режим читання даних – синхронний.

Наведемо приклад повного переліку налаштувань ETL-системи:

```python
file_path = "H:\\unzipped_data\\clients_dir\\"
if platform.system() == "Windows":
     os_file_path_delimetr = "\\"
else:
     os_file_path_delimetr = "/"
input_file_ext = file_path.split(".")[-1]
default_input_file_ext = "txt"  # if path type is dir - for filter files in dir
input_file_name        =        file_path.split(os_file_path_delimetr)[-1].split(".")[0]
input_file_encoding = "utf-8"
output_file_encoding = "utf-8"

rows_delimetr = "\n"
base_column_delimetr = "\t"
additioonal_column_delimetr = ","
output_column_delimetr = "ϬϬ"

file_read_mode = read_mode.read_full_file
rows_in_splitted_files = 1000
work_mode = work_mode.sync_work_mode
file_size_border_in_bytes = 20000000
memory_usage_politics = memory_usage_politics.setted_memory_border
path_type = path_type.is_dir
min_columns_in_row = 21
process_subcolumns_sequentially = True

units["address"].unit_is_disabled = True
raw_data_mapping_unit_dict = {
0: ([4], [units["isdn"]], True),  # bool val - need replace duplicates in columns
1: ([1, 2, 3], [units["name"]], False),
2: ([5], [units["address"]], False),
3: ([6], [units["birthdate"]], False),
4: ([11], [units["email"]], False)
}
data_concatenation_dict = {0: [0], 1: [1], 2: [2], 3: [3], 4: [4]}
```

Прокоментуємо деякі з них:

— `file_path` – повний шлях до файлу або директорії;

— `os_file_path_delimetr` – роздільник шляху до файлу, залежно від сімейства ОС (*nix або Windows) - використовується під час роботи з файлами - читання і запису - конструювання шляху вихідного файлу;

— `input_file_ext` – використовується під час роботи з файлами - читання і запису - конструювання шляху вихідного файлу або фільтрації файлів у директорії під час читання;

— `default_input_file_ext` — розширення, яке використовується, якщо `file_path` - задано як шлях до директорії, а не до файлу і закінчується на системний роздільник шляху, наприклад: `dir\\another_dir\\another_dir2\\\\` – закінчується на `\\\\` , тоді як шлях до файлу, а не директорії, - має у своєму шляху крапку, після якої йде розширення файлу. У разі, якщо вказано шлях до директорії - неможливо отримати розширення файлу під час розбивки за крапкою, у такому разі й використовується `default_input_file_ext` - як резерв - для конструювання шляху вихідного файлу або фільтрації файлів у директорії під час читання;

— `process_subcolumns_sequentially` – режим обробки суб-стовпців усередині одного стовпця. Якщо всередині одного стовпця необхідно провести розбиття значень за додатковим роздільником – у такий спосіб розділяючи один стовпчик – на безліч додаткових – існує два варіанти обробки отриманих суб-стовпців – відносно закріплених за основним стовпцем – правил.

1. Варіант «послідовної» обробки. Для кожного суб-стовпця – застосовуватимуться всі правила заданого ланцюжка. Наприклад, є два стовпці і три правила: `[стовпчик1, стовпчик2]; [правило1, правило2, правило3]` – таким чином, спершу перший стовпчик буде оброблено згідно з першим правилом, потім результат обробки буде оброблено згідно з другим правилом і т. д. – тобто вхідними даними для кожного наступного правила – є результат роботи попереднього правила, а кожний суб-стовпчик – пройде весь ланцюжок правил.

2. Маппінг суб-стовпця на відповідне правило, наприклад: `[стовпчик1, стовпчик2]; [правило1, правило2] -> [стовпчик1 - правило1;стовпчик2 - правило2]` – при цьому, обов’язковою умовою є збіг кількості суб-стовпців і заданих правил. Важливо зазначити, що цей параметр і описані для нього правила працюють саме для суб-стовпців, але не для основних стовпців.

Для більш зручної роботи і лаконічності - пропонується створити допоміжний клас, який буде зберігати в собі параметри для роботи з файлами:

```
```python
class FileObj:
```
```

```
def __init__(self, file_path: str, path_type: path_type, os_path_delimetr:
str, input_file_ext: str,
            default_input_file_ext, input_file_name: str, input_file_dir:
            str, input_file_encoding: str, output_file_encoding: str,
            file_read_mode: read_mode, rows_in_splitted_file: int,
            file_size_border_in_bytes: int, memory_usage_politics:
            memory_usage_politics,read_rows_limit:int=None,
            additional_input_file_encodings=[],standart_row_count_for_read
            =None):
      self.file_path = file_path
      self.path_type = path_type
      self.os_path_delimetr = os_path_delimetr
      self.input_file_ext = input_file_ext
      self.default_input_file_ext = default_input_file_ext
      self.input_file_name = input_file_name
      self.input_file_dir = input_file_dir
      self.input_file_encoding = input_file_encoding
      self.output_file_encoding = output_file_encoding
      self.file_read_mode = file_read_mode
      self.rows_in_splitted_file = rows_in_splitted_file
      self.file_size_border_in_bytes = file_size_border_in_bytes
      self.memory_usage_politics = memory_usage_politics
      self.read_rows_limit=read_rows_limit
      self.additional_input_file_encodings=additional_input_file_encoding
      s
      self.standart_row_count_for_read = standart_row_count_for_read```
```

І вже екземпляр цього класу передавати в конструктор об’єкта ETL-системи.

Властивість - `self.read_rows_limit` - об’єкта - `FileObj` - дає змогу здійснювати читання тільки заданої кількості рядків, що, наприклад, зручно для тестів, коли немає необхідності завантажувати весь файл:

```
```python
            with open(filepath, "r", encoding=encoding) as f:
                if os.path.isfile(filepath):
                    file_data = f.read()
                    if not read_rows_limit:
                        file_data = f.read()
                    else:
                        file_data = f.readlines()[0:read_rows_limit]
                        file_data="\\n".join(file_data)
                    return file_data
                else:
                    print("file does not exist")```
```

Властивість — `self.read_rows_limit` — об’єкта — `FileObj` — дає змогу здійснювати повторне читання файлу в інших кодуваннях, у разі виникнення помилки читання, якщо вихідне кодування - невідоме заздалегідь або може бути помилковим:
```

```python
used_encodings=[]
def  read_csv(filepath:  str,  file_size_border,  path_type:  path_type,
file_ext, encoding, default_input_file_ext,
             additional_border_s, memory_usafe_politics,
             need_create_subdir=True,
             get_previous_slice_row_c=False,
             get_previous_file_border=False,
             additional_row_c=500,
             need_filter_files_in_dir=True,read_rows_limit=None):

need_filter_files_in_dir=True,read_rows_limit=None,additional_file_encodi
ngs=[],need_filter_when_read=False):
    global ROW_C
    global used_encodings
    ........
        elif path_type == path_type.is_file:
        try:
            with open(filepath, "r", encoding=encoding) as f:
                used_encodings.append(encoding)
                if os.path.isfile(filepath):
                    if not read_rows_limit:
                        file_data = f.read()
                    else:
                        file_data = f.readlines()[0:read_rows_limit]
                        file_data="\\n".join(file_data)
                     if need_filter_when_read:
                           file_data=frozenset(file_data)
                    return file_data
                else:
                    print("file does not exist")
                    return None
        except Exception as e:
            print(e)
            exception_name=e.__class__.__name__
            if exception_name=="UnicodeDecodeError":
                if len(additional_file_encodings)>0:
                    new_file_encoding=additional_file_encodings.pop()
                    if new_file_encoding not in used_encodings:
                        used_encodings.append(new_file_encoding)

readed_data=read_csv(filepath=filepath,file_size_border=file_size_border,
path_type=path_type,file_ext=file_ext,

encoding=new_file_encoding,default_input_file_ext=default_input_file_ext,
additional_border_s=additional_border_s,

memory_usafe_politics=memory_usafe_politics,need_create_subdir=need_creat
e_subdir,

get_previous_slice_row_c=get_previous_slice_row_c,get_previous_file_borde
r=get_previous_file_border,

additional_row_c=additional_row_c,need_filter_files_in_dir=need_filter_fi
les_in_dir,
```

```
read_rows_limit=read_rows_limit,additional_file_encodings=additional_file
_encodings)
                        return readed_data
            print(traceback.format_exc())
            return None
```

Властивість - `self.standart_row_count_for_read` - об’єкта - `FileObj` - дає змогу задати кількість рядків за замовчуванням, що використовуватиметься під час розбиття файлу, більшого розміру, ніж задано властивістю `memory_usage_politics`:

```python
    if     os.path.getsize(filepath)     >     file_size_border     and
memory_usafe_politics == memory_usage_politics.setted_memory_border:
        if not get_previous_slice_row_c:
            if not standart_row_count:
                print("input slice row count:")
                row_c = input()
            else:
                row_c=str(standart_row_count)
            while not row_c.isnumeric():
                print("Incorrect  value.  Row  count  must  be  positive
integer.")
                print("input slice row count:")
                row_c = input()
            row_c = int(row_c)
            ROW_C = row_c + additional_row_c
        else:
            row_c = ROW_C
        output_dir = split_large_file(filepath, row_c, need_create_subdir)
```

1.1 Формат та використання файлів типу JSON

JSON (JavaScript Object Notation) – простий формат обміну даними, зручний для читання і написання як людиною, так і комп'ютером. Він заснований на підмножині мови програмування JavaScript, визначеної в стандарті ECMA-262 3rd Edition - December 1999. JSON - текстовий формат, повністю незалежний від мови реалізації, але він використовує угоди, знайомі програмістам С-подібних мов, таких як C, C++, C#, Java, JavaScript, Perl, Python і багатьох інших. Ці властивості роблять JSON ідеальною мовою обміну даними. JSON заснований на двох структурах даних:

а) колекція пар ключ/значення. У різних мовах, ця концепція реалізована як об'єкт, запис, структура, словник, хеш, іменований список або асоціативний масив;

б) впорядкований список значень. У більшості мов це реалізовано як масив, вектор, список або послідовність.

Це універсальні структури даних. Майже всі сучасні мови програмування підтримують їх у будь-якій формі. Логічно припустити, що формат даних, незалежний від мови програмування, має бути заснований на цих структурах.

1.2 Формати зчитування та обробки даних

За типом читання з файлу:

– читання всього файлу - повне читання;

– читання файлу по-частинах - пакетне читання;

– читання заданої кількості рядків.

За типом збереження завантажених із файлу даних даних:

– повне збереження всіх рядків - перед обробкою;

– використання структури даних `frozenset` (множина зі збереженням порядку елементів) - для попереднього фільтрування вхідних даних на наявність повних дублікатів і зменшення обсягу вхідних даних для подальшого опрацювання (параметр `need_filter_when_read`).

Формати обробки прочитаних даних:

– завантаження і цілісне опрацювання всього файлу;

– пакетне оброблення застосовується у разі:

– якщо вказаний шлях для очищення – є директорією, а не файлом, у такому разі – виконується фільтрація файлів у заданій директорії за іменем або розширенням;

– якщо розмір зазначеного файлу для очищення - перевищує заданий під час запуску системи –ліміт пам’яті (параметр `memory_usafe_politics`).

Включає в себе такі етапи:

а) розбиття файлу на частини;

б) послідовна обробка:

– очищення кожної з частин;

– збереження кожної з частин в окремий файл (для запобігання втрати даних) ;

– фінальне об’єднання всіх очищених частин - в один файл;

– збирання сміття – очищення тимчасових файлів з очищеними даними – після об’єднання у фінальний файл.

Може бути:

– повною – очищаються всі файли в списку;

– частковою – очищається тільки задана кількість файлів (параметр `self.count_of_list_part_for_proceed`).

1.3 Читання і представлення даних у пам’яті

Цей етап передбачає такі дії.

1. Розбиття прочитаних даних на рядки;
2. Розбиття кожного рядка на стовпці;
3. Обов’язкова попередня перевірка кількості елементів (стовпців) нарізаного рядка заданій константі. Якщо значення стовпців у рядку не відповідає зазначеному - рядок відкидається ще до етапу його обробки. Цей крок дає змогу гарантувати відсутність проблем з індексами в процесі опрацювання - наприклад, якщо кількість стовпців у рядку менша від очікуваної, що призведе до виходу за межі масиву або більша за очікувану, що може свідчити про зсув даних у середині рядка, якщо додаткові стовпчики не розміщено у самому кінці, що, своєю чергою, може призвести до неправильного маппінгу функцій очищення відносно даних.
4. Розбиття значень усередині стовпця за додатковим роздільником.

Крок 4 – може виконуватися як на етапі підготовки даних, так і на етапі обробки конкретного стовпця.

Робота з дампами БД. Іноді, дані можуть бути представлені не у вигляді даних у пам'яті або текстових файлів, а у вигляді дампів однієї з реляційних БД. У такому разі існує три шляхи роботи з цими даними.

1. Підключення до БД, завантаження частини даних у пам'ять і обробка їх за допомогою ETL-системи (аналогічно процесу читання з файлу.

2. Генерація SQL-запиту за допомогою параметрів, заданих під час налаштування ETL-системи, і подальший запуск цього запиту в середовищі СУБД. Запит містить у собі як умови вибірки (Select), так і умови зміни (Update) даних.

3. Комбінація перших двох варіантів - Підключення до БД, завантаження частини даних у пам'ять – на основі згенерованого SQL-запиту за допомогою параметрів, заданих під час налаштування ETL-системи, причому цей SQL-запит слугуватиме лише для вибірки даних, які надалі буде завантажено в пам'ять. Найкращий варіант для використання.

4. Варіант для дуже великих дампів 7 гб+ . Для дуже великих дампів, завантаження (імпорт) яких у СУБД - для подальшої роботи - займає надто багато часу, пропонується такий алгоритм дій:

а) розрізати файл дампа - на складові частини - як звичайний текстовий файл;

б) для кожного файлу:

– знайти рядок вигляду: `` `INSERT INTO <table_name> (field1,field2...fieldN) VALUES` ``;

– усі рядки після подібного – вважати рядками з даними, навіть якщо це рядки з іншого файлу (після розрізання великого файлу на частини);

– на підставі рядка `` `INSERT INTO` `` – скласти структуру словника, де ключі – це імена полів - `` `field1`,`field1`,`fieldN` `` - значення для ключів - значення з рядків, які вважаються рядками з даними. Таким чином – отримаємо словник значень виду `` `{filed1:value1,field2:value2,field3:value3}` ``;

– як тільки буде знайдено наступний рядок виду: `` `INSERT INTO <table_name> (field1,field2...fieldN) VALUES` `` – створити новий словник - для таблиці.

1.4 Постановка задачі

Метою кваліфікаційної роботи магістра є виконання дослідження основ та особливостей створення автономної системи обробки великих об’ємів даних. Для реалізації ПЗ необхідно виконати аналіз масивів інформації з різних предметних доменів та виокремити їх особливості для подальшого створення правил валідації та обробки вхідних значень. Для реалізації ПЗ необхідно виконати наступні кроки:

– провести аналіз методів та систем для автоматизованої обробки великих масивів даних;

– розробити принципи автоматичного розбиття та обробки текстових файлів, розмір яких перевищує доступний об’єм ОЗП;

– надати визначення та обґрунтування основних принципів та методів роботи з слабоструктурованими текстовими файлами у форматі CSV (Comma Separated Value).

Висновки до розділу 1

Одним з найважливіших етапів для обробки даних – є етап їх зчитування та першочергової підготовки, адже саме на цьому етапі буде визначено структуру даних, яка може бути змінена шляхом трансформації та такі атрибути даних – як стандарт кодування символів , базовий та додатковий роздільник атрибутів. На даному етапі також визначаються структурні та технічні обмеження, наприклад обмеження на доступний об’єм ПЗУ. При зчитуванні даних важливим показником є формат вхідного файлу – наприклад простий текстовий файл або SQL-дамп, в залежності від якого – необхідно використовувати різни техніки обробки, що подальшому впливає на швидкість роботи системи.

2 МЕТОДИ ТА АЛГОРИТМИ ОБРОБКИ ВЕЛИКИХ ДАНИХ

Процес маппінгу даних має відбуватися як на етапі читання вихідних даних, так і на етапі отримання очищених даних.

2.1 Маппінг вхідних даних

Процес маппінгу вхідних даних можна розділити на два етапи.

1. Маппінг вхідних стовпців - наприклад, три стовпці з прізвищем, ім’ям та по батькові - мають утворювати одне поле. У цьому випадку - маппінг передбачає процес з’єднання зазначених стовпців в один рядок.
2. Маппінг отриманого рядка до функцій обробки цього рядка для його очищення.

Маппінг вхідних даних – перший спосіб:

Приклад словника для маппінгу вхідних стовпців у єдине поле:

```python
columns_data_map = {
    "phone": [4, 26],
    "name": [1,2,3],
    "email": [2],
    "itn": [21]
}
```

Де ключ – ім’я сутності, а значення - індекси вхідних стовпців для кожного рядка, з яких складатиметься фінальне значення.

Приклад словника для маппінгу значень і функцій їх обробки:

```python
precessed_methods_map = {
    "phone": [validate_phone],
    "name": [validate_name],
    "email": [validate_email],
    "itn": [validate_itn]
}```

Де ключ – ім’я сутності, а значення – список функцій, які послідовно викликатимуться для опрацювання вхідного значення, до того ж для кожної наступної функції, що викликається, – її вхідним значенням – буде результат роботи попередньої.
```

Маппінг вхідних даних - другий спосіб спосіб (актуальний):

Більш удосконалений варіант цієї схеми можна представити в такому вигляді:

```
```python
raw_data_mapping_unit_dict = {
    0: ([4], [units["isdn"]], True),  # bool val - need replace duplicates
in columns
    1: ([1, 2, 3], [units["name"]], False),
    2: ([5], [units["address"]], False),
    3: ([6], [units["birthdate"]], False),
    4: ([11], [units["email"]], False)
}
data_concatenation_dict = {0: [0], 1: [1], 2: [2], 3: [3], 4: [4]}```
```

Розглянемо цей код докладніше. У цьому випадку - ми маємо два словники - `raw_data_mapping_unit_dict` і `data_concatenation_dict`.

Словник `raw_data_mapping_unit_dict` - є основним і виконує відразу дві функції.

1. Здійснює маппінг вхідних неочищених стовпців до списку etl-юнітів для їхньої обробки.

2. Здійснює попередній маппінг оброблених стовпців до попередньої структури фінальних стовпців.

Словник `data_concatenation_dict` – здійснює повторний і фінальний маппінг очищених стовпців. Цей словник необхідний у тому разі, якщо деякі стовпці необхідно обробити окремо, а потім - додатково з'єднати.

Таким чином, як можна бачити – маппінг може бути здійснено у два послідовних етапи.

1. Маппінг вхідних стовпців;

2. Додатковий маппінг уже оброблених стовпців, які пройшли маппінг на першому етапі.

Окремо відзначимо структуру словника `raw_data_mapping_unit_dict` : `0: ([4], [units["isdn"]], True), # bool val - need replace duplicates in columns :

– його ключем є індекс для попереднього маппінгу очищеного стовпця. Значення словника представлено у вигляді кортежу з трьох значень:
```

– список індексів вхідних стовпців, які необхідно обробити. Якщо в списку - більше одного індексу - перед обробкою юнітами - значення із заданих вихідних стовпців будуть склеєні в один рядок;

– список юнітів, які будуть застосовані для обробки вхідного рядка (стовпця або значення - склеєного зі значень декількох стовпців);

– булеве значення, яке вказує на необхідність видалення дублікатів у зазначених стовпцях (індексів стовпців у першому елементі кортежу) - перед їхнім об’єднанням в один рядок. Наприклад, якщо у двох стовпцях присутній номер телефону і необхідно не просто об’єднати ці два стовпці, а так само і виключити дублювання номерів. Приклад об’єднання стовпців перед викликом функцій ETL-юніта.

```
```python
if not v[2]:
          data_for_process = " ".join([columns[i] for i in v[0]])
else:
    # need remove duplicates
    data_for_process = list(set([columns[i] for i in v[0]]))
```
В новой версии системы - в случае, если не нужно удалять дубликаты - добавлена дополнительная проверка - символ, по которому нужно производить конкатенацию:
```python
if not v[2]:
    if v[1][0].concatenate_by_space and not "special_value_delimetr" in units_params[v[1][0].unit_name].keys() and units_params[v[1][0].unit_name]=="":
        concatenate_symbol = " "
    elif "special_value_delimetr" in units_params[v[1][0].unit_name].keys() and units_params[v[1][0].unit_name]!="":

concatenate_symbol=units_params[v[1][0].unit_name]["special_value_delimetr"]
    else:
        concatenate_symbol = self.additional_column_delimetr

if "additional_skip_bad_charters" in units_params[v[1][0].unit_name].keys():

     units_params[v[1][0].unit_name]["additional_skip_bad_charters"].add(self.additional_column_delimetr)
data_for_process = concatenate_symbol.join([columns[i] for i in v[0]])
```
```

Важливо: зверніть увагу на фрагмент коду:

```python
if "additional_skip_bad_charters" in units_params[v[1][0].unit_name].keys():

     units_params[v[1][0].unit_name]["additional_skip_bad_charters"].add(self.additional_column_delimetr)
```

Цей фрагмент необхідний, коли в значення юніта входить кілька стовпців, але водночас параметр перевірки на унікальність відключено (встановлений параметр False, як третє значення кортежу):

```python
1: ([17,45], [units["name"]], False)
```

У такому разі - необхідно внести список додаткового роздільника стовпця - до списку винятків, під час очищення спецсимволів (задається через параметризацію юніта)

Таку саму процедуру - треба виконувати і для символів, які можуть бути в префіксі юніта (префікс юніта - це один з його можливих параметрів, для деяких юнітів)

```python
if not self.unit_is_partically_disabled:
                if self.unit_name == "about" and units_params["about"]["info_prefix"]!="":
                    if ":" in units_params["about"]["info_prefix"]:
                        bad_charters_for_skip.add(":")
                    if "-" in units_params["about"]["info_prefix"]:
                        bad_charters_for_skip.add("-")
                         if self.data_check_funcions:
                    for f in self.data_check_funcions:
                        cleaned_data = f(cleaned_data)
                        if "additional_skip_bad_charters" in units_params[self.unit_name].keys() and len(units_params[self.unit_name]["additional_skip_bad_charters"])>0:

bad_charters_for_skip.update(units_params[self.unit_name]["additional_skip_bad_charters"])
                        cleaned_data = remove_bad_charters(cleaned_data, self.raw_data_clean_dict,bad_charters_for_skip)
```

Приклад використання параметризації для завдання додаткових символів для виключення:

```python
"about": {"info_prefix":"","additional_skip_bad_charters":set()},
```

Налаштування додаткового роздільника стовпців - для конкретного юніта. Параметр `"special_value_delimetr"` – використовується для вказівки символу-з'єднувача для декількох стовпчиків – оброблюваних юнітом, якщо стовпчики мають бути з'єднані не через пробіл або заданий під час запуску etl-конвертора – символу для об'єднання стовпчиків (`additional_column_delimetr`):

Початкова параметризація юніта:

```python
"about":
{"info_prefix":"","additional_skip_bad_charters":set(),"special_value_del
imetr":""},
```

Вказівка значення:

```python
units["about"].set_unit_params({"special_value_delimetr": "\\n"})
```

2.2 Використання технологій програмування під час маппінгу даних

Абстракцію маппінгу вхідних даних зручно представляти за допомогою концепцій ООП і класів. Для спрощення роботи з даними - можна створити клас «Юніт»:

Юніт ETL-системи – об'єкт, що має такі властивості:

– службове ім'я юніта - строкове поле - використовується на етапі налагодження та логування роботи. Наприклад, юніт, створений для опрацювання та валідації номерів мобільних телефонів, може мати службове ім'я "Phone";

– категорія перетворення даних - номери телефонів, ПІБ, посилання на веб-ресурси - для даних та інших категорій - відрізняються методи й алгоритми обробки сирих вхідних даних. Ім'я категорії, вказане під час створення об'єкта-юніта, може використовуватися як ключ до словника, який буде задано на етапі маппінгу вхідних даних (див. пункт 2 у розділі маппінг вхідних даних);

– підкатегорія в даній категорії;

– список викликаних функцій для перетворення вхідних даних - у заданій послідовності елементів списку;

– словник із даними для очищення рядка перед перетворенням.

Особливості використання програмної архітектури. Облік ключових полів. У деяких випадках, під час роботи ETL-системи - важливо враховувати такий фактор, як ключові поля. Особливість ключових полів (стовпців) усередині одного рядка - полягає в тому, що існує якась заздалегідь задана умова (комбінаторна), за якої валідним після очищення вважається той рядок, у якому присутня або певна кількість ключових полів, або їхня комбінація. Для вирішення цього завдання - існує два підходи.

1. Заздалегідь визначити список ключових ключів словника і надалі - складати з них комбінації певної довжини, наприклад:

```
```python
required_dictionary_keys=["name","phone","email"]
combinations_length=2
created_combinations=itertools.combinations(required_dictionary_keys,
combinations_length)                                                    \#
("name","phone"),("name","email"),("email","phone")
for dict_el in list_of_dicts:
     for c in created_combinations:
          if dict_el[c[0]]!="" and if dict_el[c[1]]!="":
               return true
return false
```
```

Цей метод має суттєвий недолік - за великого обсягу словників у фінальному списку та/або великого обсягу ключових полів, які дадуть довгі ланцюжки комбінацій, - повний перебір займає багато часу.

2. Вести облік ключових полів на етапі обробки - за допомогою лічильника ключових полів для кожного рядка. Під час опрацювання кожного нового рядка - лічильник ключових полів у рядках - скидається в нуль, після чого, кожен стовпчик опрацьовується відповідно до заданих правил ETL-юніту, і якщо опрацьоване значення є валідним з погляду бізнес-логіки, а ETL-юніт, за допомогою якого здійснювали опрацювання стовпчика, помічено як ключовий, тільки тоді лічильник ключових полів - збільшується. Якщо лічильник ключових полів після обробки всіх

стовпців у рядку - більший за одиницю - рядок вважається валідним. Такий підхід дає змогу додавати у фінальну множину рядків тільки валідні рядки, в яких точно присутні ключові поля, що дає змогу уникнути ситуацій, коли у фінальному рядку після обробки є не порожні стовпчики, але вони не є ключовими, і рядок не представляє цінності.

У коді це можна представити таким чином:

```python
processed_data = etl_data_unit_for_call.process_raw_data(raw_data)
if processed_data:
    processed_data = processed_data.strip()

# key unit cannot be empty, other units - can be empty
if processed_data == "" or not processed_data:
    cleaned_columns[etl_data_unit_for_call.unit_name] = ""
elif processed_data:
    cleaned_columns[etl_data_unit_for_call.unit_name] = processed_data
if processed_data and processed_data != "":
    if etl_data_unit_for_call.is_key_unit:
        key_units_count += 1
    .....
return cleaned_columns

......
if key_units_count > 1:
    cleaned_data.append(cleaned_columns)

```

Гнучкість роботи з ETL-юнітами. Основу роботи з ETL-юнітами становить заздалегідь визначений словник, що містить ETL-юніти об’єкти як значення:

```python
units = {
    "name":   data_unit("name",   data_categories.personal_data,   "name",
standart_bad_charter_list, True),
    "email":  data_unit("email",  data_categories.personal_data,  "email",
standart_bad_charter_list, True),
    "gender": data_unit("gender", data_categories.personal_data, "gender",
standart_bad_charter_list, False),
    "passport":    data_unit("passport",    data_categories.personal_data,
"passport", passport_bad_charter_list, True),```
```

Перевага заздалегідь створених об’єктів полягає в наступних наведених пунктах.

Однозначність – передбачається, що самі об’єкти не будуть змінюватися, але при цьому є можливість розширити список застосовуваних цим об’єктом методів:

```
```python [1, 2, 3], [units["name"]]```
```

Так, стовпці 1,2 і 3 - будуть оброблені ETL-юнітом – «name» (відповідно до параметра `process_subcolumns_sequentially` (див. вище)).

Сам юніт створюється таким чином:

```
`data_unit("name",          data_categories.personal_data,          "name",
standart_bad_charter_list, True)`
```

Розглянемо параметри його конструктора. Перший параметр – службове ім’я юніта. Використовується в двох цілях.

1. Як службова інформація під час логування та налагодження.
2. Як ключ для словника очищених стовпців:

```
      ```python
            if processed_data == "" or not processed_data:
                cleaned_columns[etl_data_unit_for_call.unit_name] = ""
            elif processed_data:
                cleaned_columns[etl_data_unit_for_call.unit_name]         =
processed_data
           ```
```

Другий параметр – значення класу-перерахування – категорій даних, наприклад:

```
```python
class data_categories(enum.Enum):
    personal_data = "personal_data"
    mobile_phone = "mobile_phone"
    car = "car"
    web_link = "web_link"
    geo_data = "geo_data"
    date_time_data = "date_time_data"
    other = "other"
    meta_units="meta_units"
```
```

Значення елементів цього перерахування - використовується в так само заздалегідь визначеному словнику:

```
```python
data_functions_dict = {
```

```
    data_categories.personal_data:
        {
            "name": [translate_name],
            "email": [validate_email],
            "gender": [],
            "passport": [validate_passport],
            "birthplace": [],
            "birthdate": [validate_date],
            "job_title": [validate_job_place],
            "itn": [validate_itn],
            "passport_details": [],
            "driver_license": []        },
    data_categories.mobile_phone:
        {
            "isdn": [validate_phone_number],
            "imei": [],
            "imsi": [],
            "mcc": [],
            "mnc":   []   },                        ......                  }
```

Саме використання структури вкладених словників (підкатегорій в категорію) з прив’язаними до кожної підкатегорії списками функцій - дає змогу зберігати гнучкість використання ETL-юніта шляхом розширення або скорочення списку функцій, що викликаються, до того ж самі фікції можуть (і мають) знаходитися в окремих зовнішніх файлах та імпортуватись у файл, у якому відбувається визначення цього словника, що дає змогу, абсолютно не змінюючи структури словника та самих ETL-юнітів, кардинально міняти їхню поведінку.

Третій параметр – ім’я підкатегорії в заданій категорії. Підкатегорія являє собою ключ словника в словнику заданої категорії, за яким вже відбувається отримання списку функцій для очищення (див. приклад вище). Як правило, службове ім’я юніта та ім’я підкатегорії - збігаються з логічних міркувань.

Четвертий параметр – словник символів, який застосовують під час очищення вхідних даних для цього юніта, наприклад:

```
```python
extended_bad_charter_list = [{"\\n": ["\n", "\r", "\r\n", "\n\n"]}, {"\\t":
["\t"]},
{"": [u’\u200b\u200b’, u’\xad’, u’\xa0’, ",", "+", "(", ")", "-", " ", "",
‘"‘, "‘"]},  {" ": [u’\xa0’,"""]}]
bad_charters_list_for_phone  =  [{"\\n":  ["\n",  "\r",  "\r\n",  "\n\n"]},
{"\\t": ["\t"]},
```
```

```
{"": ["Телефон", "телефон", "Тел", "тел", ".", ":", "phone", "number",
"Номер", "номер", "WhatsApp", "Viber", "Telegram"]
garbage_charter_list = [{"": ["?", "--", "__","*","!","@","-
",".",",","#","$","%","^","~","+","=","&","[","]","{","}",":","<",">","№"]}]
stop_words_list = [{"": ["НЕ ЗАДАНО", "--", "__","NULL","null"]}]
html_tag_baf_charter_list = [{"": ["<p>", "</p>",
"<strong>","</strong>"]}]
``
```

Приклад реалізації функції очищення, в якій використовується такий словник:

```
```python
def remove_bad_charters(raw_data: str, bad_charter_list: List[dict] =
standart_bad_charter_list,
                        bad_chartets_for_skip=set()):
    if raw_data:
        if isinstance(raw_data, tuple):
            if len(raw_data) > 0:
                raw_data = str(raw_data[0])
            else:
                raw_data = ""
            # value_to_str(raw_data)
        raw_data = raw_data.strip()
        if len(bad_charter_list) > 0:
            if isinstance(bad_charter_list[0], dict):
                for sublist in bad_charter_list:
                    for l_el in bad_charter_list:
                        try:
                            symb_for_remove = list(l_el.keys())[0]
                        except Exception as d:
                            pass
                        for b_ch in list(l_el.values()):
                            for b in b_ch:
                                try:
                                    #   raw_data   =   raw_data.replace(b,
symb_for_remove)
                                    try:
                                        if b not in bad_chartets_for_skip:
                                            raw_data                    =
raw_data.replace(b, symb_for_remove)
                                    except Exception as e:
                                        print(e)
                                except Exception as e1:
                                    print(e1)
                return raw_data.strip() if raw_data else ""
            else:
                return remove_bad_charters(raw_data, bad_charter_list[0],
bad_chartets_for_skip)
        return raw_data.strip() if raw_data else ""
    else:
        return ""
```
```

Можливість додавання довільної функції до юніта:

```
```python
def add_func_to_unit_list(self,func,add_type:add_func_type):
    if add_type==add_func_type.add_to_end:

data_functions_dict[self.data_category][self.unit_name].append(func)
    if add_type == add_func_type.add_to_start:

data_functions_dict[self.data_category][self.unit_name].insert(0,func)
```
```

Приклад використання:

```
```python
units["about"].add_func_to_unit_list(strip_about_data,add_func_type.add_t
o_start)```
def strip_about_data(data:str):
    try:
        return data.split(" ")[0].strip()
    except Exception as e:
        return ""
```
```

Таким чином, зазначену функцію буде додано до словника `data_functions_dict` і використано під час обробки юнітом.

Узагальнена структура роботи юнітів. Основна ідея полягає в наступному:

1. Юніт створюється один раз у зовнішньому файлі і залишається незмінним. Для роботи з юнітом - відбувається його імпорт і виклик за заданим ключем у словнику юнітів.

2. Словник юнітів залишається незмінним і знаходиться в зовнішньому файлі, що забезпечує читабельність решти коду.

3. При своєму створенні – юніт спирається на поняття категорії та підкатегорії.

4. Усі категорії та їхні підкатегорії жорстко визначені та заздалегідь задані у вигляді ієрархічної структури словників – у зовнішньому файлі.

5. Єдиний змінюваний елемент ETL-юнітів – це список функцій, що викликаються для заданої підкатегорії. Це дає змогу змінювати тільки кількість елементів у списку для заданої підкатегорії - не зачіпаючи логіку класу юніта або початковий словник створених об’єктів-юнітів.

Загальна структура викликів: виклик etl-юніта за ключем -> звернення до константного словника заздалегідь створених юнітів -> звернення до константного ієрархічного словника категорій та їхніх підкатегорій -> отримання списку функцій, які треба викликати для даної категорії та підкатегорії (і відповідно - під час використання цього юніта)

Рисунок 1.1 – Приклад ланцюга виклику функції для ETL-юніта

Переваги класу-юніта – перед іншими структурами даних. Слід одразу зазначити перевагу використання об’єкта-юніта - перед використанням звичайного словника:

```
```python
precessed_methods_map = {
    "phone": [validate_phone],
    "name": [validate_name],
    "email": [validate_email],
    "itn": [validate_itn]
}
```

– використання словника не дає змоги здійснювати облік ключових полів - без додаткових допоміжних структур даних (списків або інших словників);
```

– використання словника не дає можливості одразу вказати словник небажаних символів, які необхідно використовувати для цього стовпця (без додаткових допоміжних структур даних (списків або інших словників));

– використання словника - без дворівневої структури прив’язки (словник юнітів і словник категорій) - не дає запровадити чітке розмежування даних на категорії, що ускладнює роботу з кодом під час роботи з великою кількістю стовпців (сутностей).

Можливість вимкнення юніта. У класі юніта - присутнє поле - `unit_is_disabled` - за замовчуванням встановлене в `False`: `self.unit_is_disabled=False`. Це поле дає змогу "вимкнути" юніт перед його використанням, перед прив’язкою його до стовпця, що зі свого боку дає змогу зробити перелічені нижче дії.

1. Зберегти вказаний стовпець незмінним, тобто не піддавати його обробці - якщо це необхідно.

2. Продовжувати використовувати службове ім’я юніта як ключ для словника очищених стовпців - під час обробки даних:

```
```python
if etl_data_unit_for_call and not etl_data_unit_for_call.unit_is_disabled:
    \# произвести очитску данных с помощью юнита
else:
    cleaned_columns[etl_data_unit_for_call.unit_name] = raw_data
return
```
```

Приклад «вимкнення» юніта - перед використанням:

```
```python
units["address"].unit_is_disabled = True
raw_data_mapping_unit_dict = {
    0: ([4], [units["isdn"]], True),  # bool val - need replace duplicates in columns
    1: ([1, 2, 3], [units["name"]], False),
    2: ([5], [units["address"]], False),
    3: ([6], [units["birthdate"]], False),
    4: ([11], [units["email"]], False)
}
```
```

Оскільки всі об’єкти юнітів створені заздалегідь і зберігатимуться в словнику, для «вимкнення» юніта необхідно звернутися до словника за потрібним ключем і встановити значення поля `unit_is_disabled` - у юніта - у значення `True`.

Часткове вимкнення юніта. Цей варіант вимкнення юніта дає змогу залишити значення стовпця незмінним - не рахуючи видалення символів зі списку поганих символів.

Так само важливо зазначити, що за великої кількості стовпців у вхідному файлі (а відповідно і юнітів - для очищення) - незручно вручну виставляти параметр часткового вимкнення - для кожного юніта. Саме тому, доцільно передбачити прапор часткового вимкнення для всіх юнітів, який, якщо він встановлений, буде задавати значення цього параметра - для кожного юніта в тілі конструктора об’єкта ETL-системи. Це дасть змогу вимкнути всі юніти і ввімкнути - тільки необхідні - одиничні, тим самим - інвертувавши логіку.

Параметризація юніта. Для юніта додано функцію такого вигляду:

```
```python
def set_unit_params(self,params:dict):
    accesed_unit_params=list(self.unit_params_dict.keys())
    for k,v in params.items():
        if k in accesed_unit_params:
            self.unit_params_dict[k]=params[k]
```
```

Приклад використання функції:

```python
units["birthdate"].set_unit_params({"date_format": ""})
```

Ця функція приймає як параметр - словник, де ключ - ім’я параметра що встановлюється та значення - значення цього параметра. Якщо ключі в переданому словнику - з піску дозволених параметрів для даного юніта - значення з переданого у функцію словника - встановлюються як значення словника налаштувань юніта, які потім, своєю чергою, використовуються у функції перевірки та обробки даних.

Тепер, створення юніта - має такий вигляд:

```python
"birthdate":     data_unit("birthdate",     data_categories.personal_data,     "birthdate",
standart_bad_charter_list, "array",units_params["birthdate"],False),
```
- відбувається звернення до словника unit_params - який, своєю чергою, - так само містить словники як значення, які потім і використовуються у функції `set_unit_params` :
```python
units_params = {
    "name": {},
    "email": {},
    "passport": {},
    "birthplace": {},
    "birthdate": {"date_format": ""},
....
}
```

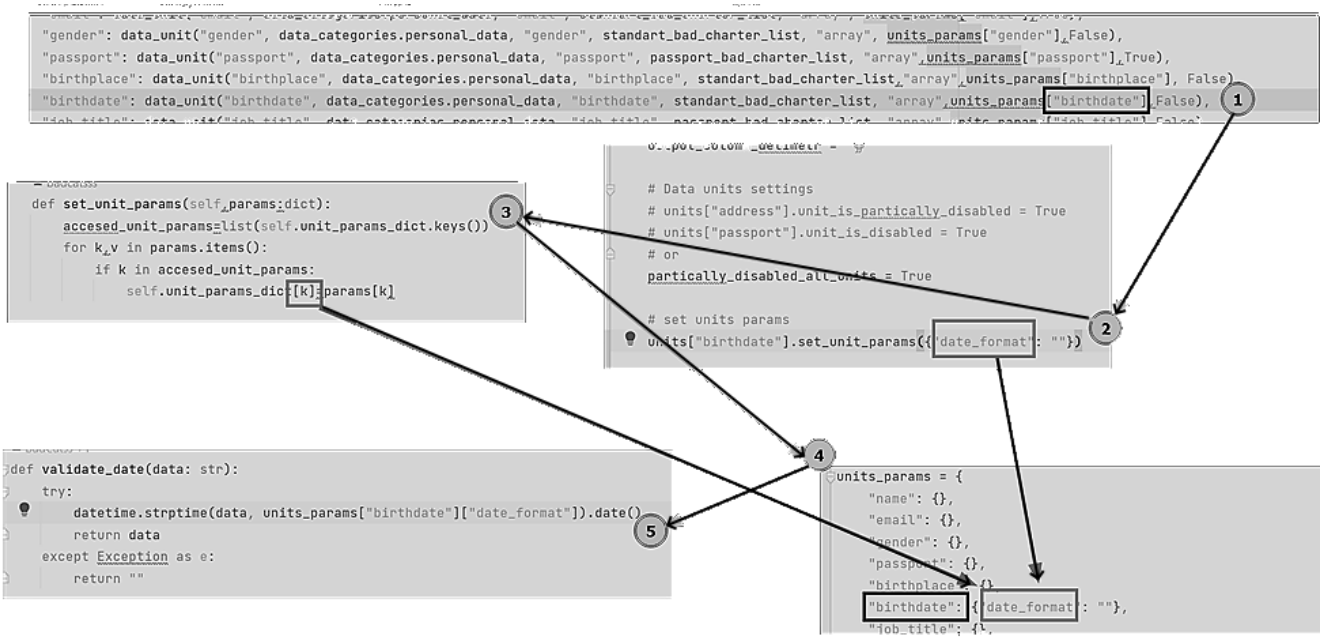

Рисунок 1.2 – Приклад послідовності параметризації юніта

Як можна бачити - на скріншоті вище - при створенні юніта йому одразу встановлюється заздалегідь зумовлений словник дозволених параметрів (`unit_params`) - в ключах якого потім і відбувається перевірка в методі `set_unit_params` - якщо ключ знайдений - параметру встановлюється задане значення, а самі параметри та їхні значення - вже використовуються у функціях перевірки, як, наприклад, це показано для функції перевірки формату дати.

Операції зі стовпцями. Поділ рядка - на групу стовпців:

```python
raw_data_mapping_unit_dict = {
    1: {  # columns group
        0: ([1, 2, 3], [units["name"]], False),  # bool val - need replace duplicates in columns
        1: ([7], [units["email"]], False),
        2: ([11, 12, 13, 14], [units["address"]], False),
        3: ([19], [units["passport"]], False),
        4: ([20, 21], [units["isdn"]], True),
        5: ([22], [units["gender"]], False),
        6: ([23], [units["education"]], False),
        7: ([37], [units["birthdate"]], False),
        8: ([87], [units["social_url"]], False)

    },
    2: {  # columns group
        0: ([41, 42, 43], [units["name"]], False),  # bool val - need replace duplicates in columns
        1: ([-1], [units["email"]], False),
        2: ([50, 51, 52, 53], [units["address"]], False),
        3: ([-1], [units["passport"]], False),
        4: ([54, 55], [units["isdn"]], True),
        5: ([60], [units["gender"]], False),
        6: ([-1], [units["education"]], False),
        7: ([45], [units["birthdate"]], False),
        8: ([88], [units["social_url"]], False)
    }
}
```

Цей варіант може використовуватися у випадку, коли - в одному рядку присутні повторювані типи стовпців з різною інформацією, наприклад інформація про двох співробітників або два пов’язані товари. У такому разі можна говорити про формальний поділ рядка на дві групи стовпців, які мають оброблятися за еквівалентними правилам і на виході - кожна група стовпців - має створювати окремий фінальний рядок. Приклад початкового рядка:

```
`employee_1_name,employee_1_name2,employee_1_surname,employee_1_address,employee_1_passport,employee_1_phone1,employee_1_phone2,employee_1_education,employee_1_birthdate,employee
```

```
_1_social_network,employee_2_name,employee_2_name2,employee_2_surname,employee_2_address,
employee_2_passport,employee_2_phone1,employee_2_phone2,employee_2_education,employee_2_b
irthdate,employee_2_social_network`
```

Як можна бачити, у початковому рядку - присутні поля - `name,name2,surname,address,passport,phone1,phone2,education,birthdate,social_network` , при цьому дана група полів – присутня двічі – для `employee1` і `employee2` відповідно, необхідно обробити ці дві групи за однаковими правилами й отримати два окремі рядки.

Так само, слід зазначити, що для індексів деяких вхідних стовпців в одній із груп встановлено значення -1, що свідчить про необхідність вставити на це місце у фінальному рядку - пропуск (порожній рядок). Це робиться для того, щоб дотримуватися однакової кількості стовпців під час роботи з різними групами всередині одного рядка, оскільки кількість стовпців між групами може не збігатися в деяких випадках.

Ця модифікація не вимагає особливих змін у коді - приклад коду без використання груп:

```
```python
try:
            for row in data_for_clean:
                                    cleaned_columns = dict()
                key_units_count = 0
                columns = row.split(self.base_column_delimetr)
                if len(columns) == self.min_columns_in_row:
                     for v in self.raw_data_unit_mapping_dict.values(): \# получаем кортежи
                             ........
```
```

Приклад коду з використанням груп стовпців:

```
```python
try:
            for row in data_for_clean:

                columns = row.split(self.base_column_delimetr)
                if len(columns) == self.min_columns_in_row:

                    for  v1  in  self.raw_data_unit_mapping_dict.values():  \#  получаем
словарь-группу столбцов
                        cleaned_columns = dict()
                        key_units_count = 0
                        if isinstance(v1, dict):
                            for v in v1.values(): \# получаем кортежи
                            ........
```
```

```
```

Основна відмінність полягає в тому, що створення нового словника очищених стовпців і лічильника ключових полів відбувається - у другому випадку не тільки для кожного рядка, а й для кожної групи стовпців у рядку.

Ту саму ідею поділу рядка на групу стовпців - можна продемонструвати і без прив'язки до ідеї юнітів:

```python
def split_cols_to_new_rows(cols,max_cols_count,separator_symb):
    new_lines_for_add = []
    if len(cols)>max_cols_count:
          for i in range(0, len(cols), max_cols_count):
              x = i
              print(cols[x:x + max_cols_count - 1])
              c = cols[x:x + max_cols_count - 1]
              while len(c) < max_cols_count:
                  c.append("")
              new_lines_for_add.append(separator_symb.join(c))
      return new_lines_for_add
```

Поділ одного стовпця - на кілька юнітів

```python
raw_data_mapping_unit_dict = {
    1: {  # columns group
        0: ([1, 2, 3], [units["name"]], False),  # bool val - need replace duplicates in columns
        1: ([7], [units["email"]], False),
        2: ([11, 12, 13, 14], [units["address"]], False),
        3: ([19], [units["passport"]], False),
        4: ([20, 21], [units["isdn"]], True),
        5: ([22], [units["gender"]], False),
        6: ([23], [units["education"]], False),
        7: ([37], [units["birthdate"]], False),
        (8,9):    ([87],    [units["passport"],units["passport_details"]], False)
    }
}
```

У прикладі коду вище, можна побачити, що стовпчик 87 - в оригінальних даних - містить у собі два види інформації, яку в системі можна подати як окремі стовпчики – «passport» і «passport_details». У такому разі, як індекс (ключ словника)

- вказується кортеж індексів. Кількість елементів у кортежі - має збігатися з кількістю юнітів у списку для опрацювання, які йдуть послідовно в тому порядку, в якому йдуть дані в оригінальному стовпчику, - розділені через додатковий роздільник. У цьому прикладі, в оригінальному стовпчику спершу йде інформація про паспорт, а потім - додаткові відомості про нього; Оригінальний стовпчик буде розбитий за додатковим роздільником, і кожна частина - буде оброблена відповідним юнітом зі списку та додана за відповідним індексом з кортежу індексів - як стовпчик до очищеного файлу. Тобто частина оригінального стовпця - в якій міститься інформація – «passport» - буде додана як 8 стовпець в очищеному файлі, а частина, в якій міститься інформація – «passport_details» - як 9 стовпець.

Видалення дублікатів. Типи перевірок значень для видалення дублікатів.

– горизонтальна перевірка - має на увазі порівняння стовпців - у рамках одного рядка - на наявність значень, що дублюються;

– вертикальна перевірка - рядкова перевірка значень - за конкретним стовпцем. Якщо значення в стовпці або стовпцях збігається у двох рядках - рядки вважаються дублікатами. За наявності декількох стовпців для перевірки - у разі збігу значень хоча б по одному з них у двох рядках - вважається дублікатом;

– об’єднання вертикальної та горизонтальної перевірки - має на увазі вертикальну перевірку за умови, що значення збігається для всіх перерахованих для пошуку дублікатів стовпців. За наявності декількох стовпців для перевірки - рядок вважається дублікатом, тільки якщо всі задані стовпці в різних рядках (вертикальна перевірка) - збіглися між собою (горизонтальна перевірка).

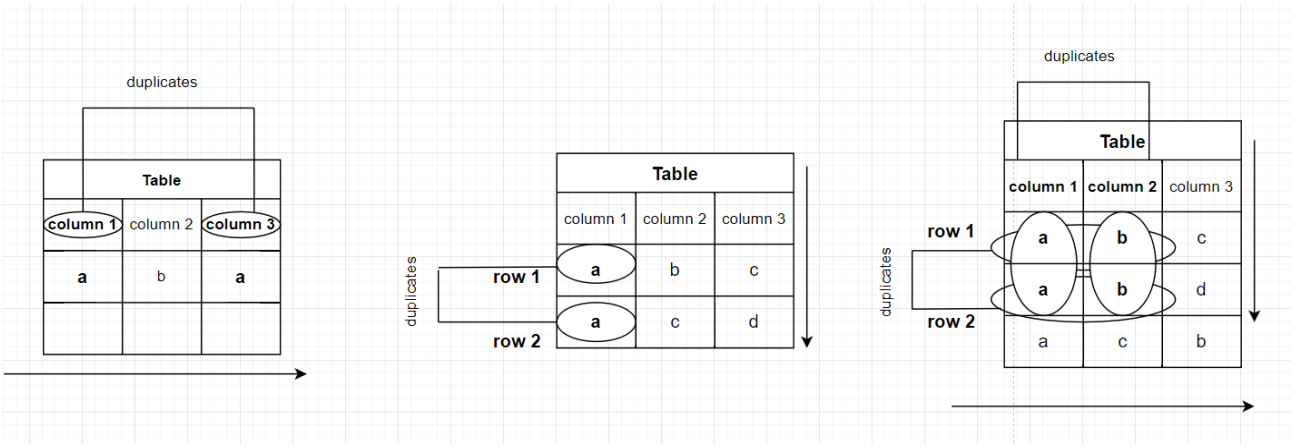

Рисунок 1.3 – Види дублікатів даних

Типи видалення дублікатів

Видалення дублікатів можна розділити на такі категорії:

– видалення повних дублікатів рядків - на етапі читання даних;

– видалення дублікатів за стовпцями - на етапі обробки:

– основне видалення дублікатів з об’єднанням значень стовпців рядків за ключовим полем, що добирається (включає в себе сортування рядка і пошук донизу - дубльованих «спів сполук» з об’єднанням значень з ними і позначкою їх - як дублікати);

– вторинне видалення дублікатів за ключовим полем, що добирається, - без об’єднання значень стовпців (включає в себе словник - з ключами категоріями - `name`, `email`, `isdn`, які всередині себе так само містять словники - де ключ - це значення цього поля в категорії, наприклад, номер телефону, а значення за цим ключем - це ПОВНИЙ об’єкт словника, в якому цей ключ для даної глобальної категорії міститься. Такий підхід дає змогу виконувати швидкий пошук у НАЙБІЛЬШІЙ категорій ключів - глобальній категорії, не роблячи сортування і порівнюючи тільки два сусідні об’єкти).

Словник категорій (name,isd,email) –> словник конкретної категорії - > ключ - значення, що належить до даної категорії: значення - повний об’єкт словника, що містить даний ключ`

Перевірка дублікатів шляхом хешування - перед записом до файлу (перевірка хеша рядка у файлі з хешами - раніше записаних рядків)

Об’єднання рядків «дублікатів» за ключовим полем (Основне видалення дублікатів з об’єднанням значень стовпців рядків за ключовим полем, що добирається).

Дані після очищення можуть бути відфільтровані такими способами.

1. Просте видалення рядків, що повністю дублюються, - для цього вистачить використання стандартної структури даних - "set".

2. Об’єднання двох рядків - "стовпці до стовпця" - у разі знаходження спільного ключового стовпця (стовпця, в якому значення для даних рядків - однакові). Варто зазначити, що до вибору такого стовпця необхідно підходити відповідно - ним може бути, наприклад, адреса електронної пошти або номер паспорта - тобто такі стовпчики, які, по-визначенню, є унікальними в межах одного набору даних. Розглянемо другий випадок.

Під час об’єднання даних за ключовим стовпчиком - виникає проблема того, що такі рядки можуть перебувати на досить великій один від одного відстані, що веде до проблеми порівняння кожного рядка з кожним нижчим. Для розв’язання цієї проблеми та оптимізації роботи - необхідно впорядкувати рядки за шуканим ключовим стовпчиком, тим самим - розташувавши рядки з однаковими стовпчиками поруч, що дасть змогу перевіряти подібність рядка лише з одним рядком, що знаходиться нижче (тобто N+1). У разі знаходження рядка, який дублює ключове поле, дані стовпців цього рядка об’єднуються з даними відповідних стовпців цього рядка, а нижчий рядок, який був дублем, видаляється зі списку рядків.

```
Приклад реалізації такої функції:
```python
```
```

```
def merge_duplicates(self,cleaned_data,merge_by_filed:str):
        if len(cleaned_data)>0 and merge_by_filed != "":
            # sort
            if merge_by_filed in cleaned_data[0].keys():
                cleaned_data=sorted(cleaned_data,      key=lambda      d:
d[merge_by_filed])

                for i in range(0,len(cleaned_data)-1):
                    if
cleaned_data[i][merge_by_filed].lower()==cleaned_data[i+1][merge_by_filed
].lower()
                        if  cleaned_data[i][merge_by_filed]  !=  ""  and
cleaned_data[i][merge_by_filed] != "\n":
                            keys=list(cleaned_data[i].keys())
                            keys.remove(merge_by_filed)
                            for k in keys:

cleaned_data[i][k]=self.additional_column_delimetr.join(set([cleaned_data
[i][k],cleaned_data[i+1][k]]))
                            cleaned_data.remove(cleaned_data[i+1])
```

Так само, важливо розуміти, що не всі стовпці необхідно буде склеювати при знаходженні дублів, наприклад, якщо є рядок з полями - ПІБ, e-mail, номер телефону, і ми хочемо використовувати `e-mail` - як ключове унікальне поле, за яким будемо виконувати порівняння рядків, то при знаходженні дублікатів нам, скоріш за все, не потрібно буде склеювати в одному результуючому рядку значення стовпчиків з ім’ям, на відміну від номерів телефону. Для вирішення цієї проблеми - необхідно заздалегідь визначити список полів, які потрібно пропускати:

```python
merge_duplicates_skip_merge_columns = ["name", "address", "passport"]
```

```python
if k not in merge_duplicates_skip_merge_columns:
```

При склеюванні стовпців в один - так само можливі такі варіанти:

1. Просте об’єднання стовпців через роздільник;
2. розбивка кожного стовпчика за вказаним роздільником, перетворення їх на множини, об’єднання двох множин і, нарешті, склеювання елементів

результуючої множини - через роздільник - у фінальний рядок - значення стовпчика.

Перший варіант підійде для простих значень, які не потрібно перевіряти на унікальність - наприклад, стовпці з адресою проживання (так, вона теж може збігатися, але їх потрібно порівнювати між собою як цілісні рядки, а не як дві множини (оскільки після розбивки рядків на множини - їхні елементи - не зберігають порядок і неможливо відновити цілісний рядок; навіть у разі використання структури даних "впорядкована множина" (frozenset) - під час об'єднання цих двох множин - їхні елементи збережуть порядок, але в разі видалення дублів та додавання нових елементів із другої множини - фінальний рядок - однаково втратить свій сенс, як це можна показати на прикладі двох рядків з адресою:

`вул. Б. Морська будинок 21` і `Вул. Велика Морська будинок 21` - що в підсумку - дасть нам рядок: `вул. Вул. Б. Велика Морська будинок 21`)), а другий варіант підходить для фільтрації значень у стовпчику, наприклад, під час об'єднання стовпців із номерами мобільних телефонів.

Так само, під час склеювання рядків за стовпчиками - важливо враховувати, що стовпчики не прийняли порожнє значення - тобто не дорівнюють рядку - `""` і не дорівнюють символу перенесення рядка - `\n` - тому що, якщо допустити такий варіант, - т. к. трапиться така структура стовпчиків в очищених даних, коли в словниках кожного рядка - у стовпчику, що буде вказано як ключовий для очищення дублів - буде або порожній рядок, або символ перенесення рядка, то з дуже великою вірогідністю - буде склеєно більшість рядків. Для запобігання такій ситуації і відповідає дві перевірки:

```python
len(cleaned_data)>0 and merge_by_filed != "" and merge_by_filed != " "

cleaned_data[i][merge_by_filed] != "" and cleaned_data[i][merge_by_filed] != "\n" and merge_by_filed != " ":
```

Типи пропусків значень – при об’єднанні дублікатів.

При об’єднанні значень стовпців при дублюючих рядках - стовпці можна умовно розділити на дві категорії:

– стовпці - за якими виконується перевірка дублікатів рядків;

– стовпці - значення яких буде об’єднано в разі збігу значень стовпців - першого типу.

Можливі такі варіанти пропуску об’єднання стовпців під час пошуку дублікатів:

1. Пропуск стовпців - для об’єднання. Використовується список стовпців – `merge_duplicates_skip_merge_columns`. Значення стовпців, зазначених у цьому списку - не об’єднуються через роздільник у разі дублікатів за стовпцями, які використовуються для перевірки дублікатів. Використовується значення першого рядка з двох, значення стовпця другого рядка - відкидається. При використанні даного типу пропуску - об’єднання значень дублікатів рядків - не проводиться зовсім. **Даний тип пропуску - ігнорує стовпець для об’єднання.**

2. Пропуск стовпців - для об’єднання - за значенням стовпця. При використанні цього типу пропуску - стовпець - не ігнорується для об’єднання - об’єднання значень стовпців - для дублікатів рядків - проводиться, якщо саме ЗНАЧЕННЯ стовпця - не внесено до чорного списку. **Цей тип пропуску - ігнорує значення стовпця для об’єднання**.

Розглянемо граничний випадок, коли ключове поля для об’єднання дублікатів - порожнє (дорівнює - `""""`) або дорівнює символу переносу рядка, або значення стовпчика - для об’єднання (але не сам стовпчик) - перебуває в чорному списку (дивись докладніше - "типи об’єднання стовпчиків"):

```
```python
if cleaned_data[i][merge_by_filed] == cleaned_data[i + 1][merge_by_filed]:
                                                        if
cleaned_data[i][merge_by_filed] not in self.duplicates_merge_values_skip
and        cleaned_data[i][merge_by_filed]        !=        ""        and
cleaned_data[i][merge_by_filed] != "\n":
                                    self.replace_duplicate_value(cleaned_
data,                                                                    i,
merge_by_filed,need_filter_then_merge,merge_duplicates_skip_merge_columns
,
                                                                 need_add
itional_split_by_space)

                                   merge_by_filed = merge_by_filed_base
```
```

```
                                else:
                                    for uniq_filed_for_check_duplicates in
self.additional_remove_duplicates_by_column_key:
                                                                    merge_by_filed  =
uniq_filed_for_check_duplicates
                                                self.replace_duplicate_value(clea
ned_data,                                                                       i,
merge_by_filed,need_filter_then_merge,merge_duplicates_skip_merge_columns
,
                                                                            need
_additional_split_by_space)
                                            merge_by_filed = merge_by_filed_base
```

Саме для того випадку, було додано додаткову гілку `else` - яка в разі свого спрацьовування - проходиться циклом за додатковим списком полів, які можуть використовуватися для перевірки рядків на дублікати. При цьому, оскільки ключове поле для перевірки - змінюється при кожній ітерації - необхідно так само, не забувати - повертати йому значення основного

Логіку роботи методу - `replace_duplicate_value` - наведено нижче.

1. Створюється дві додаткові структури даних.
2. Словник `founded_duplicates_for_uniq_fields` - словник, множин, у які додаються знайдені значення для дублікатів (тобто якщо спрацювала умова

```python
 (cleaned_data[i][merge_by_filed] == cleaned_data[i + 1][merge_by_filed]
and        cleaned_data[i][merge_by_filed]        !=        ""        and
cleaned_data[i][merge_by_filed] != "\n")
```

Та умова:

```python
if "FOR REMOVE" not in cleaned_data[i].values() and "FOR REMOVE" not in
cleaned_data[i + 1].values():
```

Яке свідчить про те, що дублікат не був знайдений у попередньому рядку (див. докладніше - нижче)):

Цей словник - має такий вигляд:

```python
```

```
‘email’:("test_email1@gmail.com","test_email2@gmail.com")
‘phones’:("123456","789101112")
```

Словник `founded_duplicates_for_uniq_fields_indexes` - словник, який містить пару - значення унікального поля для перевірки, що було додано до словника `founded_duplicates_for_uniq_fields` та індекс рядка, у якому значення цього стовпця зустрічається. Словник має такий вигляд:

```python
"test_email1@gmail.com":23,
"789101112":89
```

Після того, як були надані відповідні пари ключ-значення - для обох словників - значення для наступного рядка - замінюються на заглушку:

```python
for k in keys:
     cleaned_data[i + 1][k] = "FOR REMOVE"
```

Варто окремо пояснити, чому значення замінюються на заглушку, а не видаляється весь об’єкт словника зі списку рядків (списку словників). Річ у тім, що перебуваючи всередині циклу і намагаючись видалити елемент списку, ми порушуємо порядок індексів, що надалі призводить до виходу за межі масиву.

Як можна бачити з прикладу – ім’я збіжиться для всіх 3х рядків, але тому що ім’я не є ключовим полем - ми не можемо спиратися на нього - для того, щоб об’єднати відповідні стовпці рядків. Так само видно, що в перших двох рядках - збігається номер мобільного телефону, для третього рядка - він відсутній. Для рядка 1 і 3 - збігається адреса електронної пошти.

Візьмемо за базове поле - номер телефону і розглянемо роботу алгоритму по кроках:

Перевіряємо умову:

```python
(cleaned_data[i][merge_by_filed] == cleaned_data[i + 1][merge_by_filed] and cleaned_data[i][merge_by_filed] != "" and cleaned_data[i][merge_by_filed] != "\n")
```

Якщо умова з пункту 1 - виконується - перевіряємо умову:

```python
if "FOR REMOVE" not in cleaned_data[i].values() and "FOR REMOVE" not in cleaned_data[i + 1].values():
```

Якщо умова - виконується - об’єднуємо два словники – викликавши метод merge_two_dict.

Вносимо знайдене дубльоване значення за ключем - ім’ям ключового поля - у словник `founded_duplicates_for_uniq_fields`:

```python
"phones":(12345),
"email":("some_email1")
```

Вносимо знайдене дубльоване значення - як ключ до словника - `founded_duplicates_for_uniq_fields_indexes` та індекс рядка, в якому воно було зустрінете, **але використавши значення з першого рядка, а не з рядка - дубліката (другого)**:

```python
"12345":0,
"some_email1":0
```

Оскільки ми зустріли це значення в першому рядку - вказуємо індекс - нуль - як значення (у цьому прикладі).

Зверніть увагу, що в обидва словники - ми вносимо не тільки саме ключове поле, що дублюється, а й усі поля **першого** із двох дублікатів рядків - які будуть замінені на заглушку (див. пункт 2 і 6 - для деталей).

Замінюємо значення в словнику другого рядка (який дублює перший) - на значення заглушки:

```python
"name":"FOR REMOVE",
"email":"FOR REMOVE"
"phone":"12345",
```

Список словників - набув такого вигляду:

```
person1;some_email1;12345
FOR REMOVE;FOR REMOVE;12345
person1;some_email1;
```

Перейти до першого кроку. Якщо на попередньому рядку - є значення - `FOR REMOVE` - це означає, що цей рядок уже було об’єднано з іншим (не обов’язково з попереднім - тому що може бути кілька таких рядків поспіль, які необхідно об’єднати з найпершим рядком). У такому разі, необхідно об’єднати поточний рядок (у нашому прикладі - це рядок 3) - з найпершим рядком, де зустрічається значення цього ключового поля. Саме для знаходження індексу цього рядка і використовується словник - `founded_duplicates_for_uniq_fields_indexes` - в якому ми шукаємо за значенням ключового поля - індекс рядка, з яким потрібно об’єднати - цей.

Проблема полягає в тому, що, як можна бачити в даному прикладі - для третього рядка - ключове поле з номером телефону - виявляється порожнім. У такому разі - необхідно вдатися до згаданого раніше списку - `additional_remove_duplicates_by_column_key` (у наведеному вище коді - циклічно циклічно проходячись цим списком, - ми викликаємо метод `replace_duplicate_value`), один з елементів якого, використаний як ключове поле для

перевірки - об’єднання рядків - може бути спершу в словнику `founded_duplicates_for_uniq_fields`, а потім у словнику - `founded_duplicates_for_uniq_fields_indexes`, що дасть змогу так само об’єднати відповідні рядки.

Тобто в цьому прикладі, об’єднавши рядки 1 і 2 - за збіжним полем номера телефону, ми одразу ж внесли у відповідні словники ще й значення адреси електронної пошти **першого рядка з двох** дублікатів, що дало надалі змогу з’єднати рядки 1 і 3, але вже за значенням поля адреси електронної пошти (тому що значення поля номера телефону - було відсутнє для рядка номер 3).

Можливі такі варіанти - при об’єднанні рядків:

– рядки рівні - об’єднання не потрібне - можна повернути будь-який із рядків (НЕ виконується умова `if value_by_key!=second_value_by_key`);

– рядки не рівні й обидва рядки валідні (Виконується умова `if value_by_key!=second_value_by_key:`);

– один рядок - є підмножина другого - повернути другий рядок (довжина симетричної множини - дорівнює нулю: НЕ виконується умова: `if len(string_diff)>0`);

– один рядок НЕ є підмножина другого - об’єднати рядки (довжина симетричної множини - НЕ дорівнює нулю: значення складових рядка не перетинається повною мірою, щоб вважати один із рядків повним підрядком другого);

– рядки не рівні, але один із рядків - не валідний (у нашому випадку - один із рядків дорівнює значенню-заглушці - `FOR REMOVE`) - повернути тільки валідний рядок (НЕ виконується умова - `if value_by_key!="FOR REMOVE" and _value_by_key!="FOR REMOVE"`).

Важливо: параметр `need_filter_then_merge` - вказує - чи повернути об’єднання різниці (об’єднання рядків, виключаючи перетин), чи просто повернути об’єднані між собою рядки.

Як «синтаксичний цукор» - у систему додано можливість швидкого заповнення списку `merge_duplicates_skip_merge_columns` :

```python
if len(merge_duplicates_skip_merge_columns)==1 and "*" in merge_duplicates_skip_merge_columns:
    merge_duplicates_skip_merge_columns+=list(cleaned_data[0].keys())
    merge_duplicates_skip_merge_columns.remove("*")
    merge_duplicates_skip_merge_columns.remove(merge_by_filed)
```

Якщо в список `merge_duplicates_skip_merge_columns` - передано символ `*` - всі юніти будуть автоматично додані в список для пропуску при об’єднанні значень

Видалення дублікатів перед записом у файл (постпроцессинг пошуку дублікатів).

Вторинне видалення дублікатів за ключовим полем, що добирається, - без об’єднання значень стовпців:

– перевірка всіх відсортованих значень за заданим ключовим полем (`remove_duplicates_by_column`);

– перевірка всіх відсортованих значень за окремим списком полів - для видалення дублікатів перед записом у файл (`postprocessing_duplcates_remove_keys`).

```python
if d1[etl_convertor.remove_duplicates_by_column]==d2[etl_convertor.remove_duplicates_by_column] and d1[etl_convertor.remove_duplicates_by_column]!='':
            data[i]="FOR REMOVE"

for d_k in etl_convertor.postprocessing_duplcates_remove_keys:
            if d_k in d1.keys() and d_k in d2.keys():
                if d1[d_k] == d2[d_k] and d1[d_k] != '':
                    data[i] = "FOR REMOVE"
headers=data[0]
data=list(frozenset(data[1:]))
data.insert(0,headers)
if "FOR REMOVE" in data:
    data.remove("FOR REMOVE")
with open(file_path, "w", encoding=output_file_encoding) as f:
    f.writelines(data)
```

Основна відмінність оновленого постпроцесингу пошук дублікатів від основного пошуку дублікатів - полягає у **відсутності сортування списку значень за зазначеним стовпцем** . Весь пошук дублікатів на етапі постпроцесингу полягає в порівнянні **тільки двох рядків** - і і і+1. Основна суть такого пошуку полягає в такому:

Розглянемо рядок:

```python
merged_keys_values_dict[d_k][d1[d_k]]=(i,d1)
```

Де `d1` - словник зі значеннями стовпців поточного рядка, що перевіряється на дублікати, наприклад:

```python
d1={
       "isdn":"+12345678",
       "email":"test123@mailbox.com"}
```

і список ключів для перевірки дублікатів - `etl_convertor.postprocessing_duplcates_remove_keys` - наприклад такий: `[isdn,email]`
- `merged_keys_values_dict`

Це словник, що має вигляд:

```python
{
"isdn":{
"+12345678":(1,{
       "isdn":"+12345678",
       "email":"test123@mailbox.com"
      })
},
"email":{
"test123@mailbox.com":
      (1,{
       "isdn":"+12345678",
       "email":"test123@mailbox.com"
      })}}
```

Тобто кожне значення за ключовими полями - для кожного рядка файлу, що переглядається, - заноситься до словника - до категорії ключового поля (тобто окремий під-словник для ключового поля «isdn» та окремий - для «email») - куди, своєю чергою, заносяться словники, в яких ключ - безпосередньо значення цього ключового поля для поточного рядка, а значення - кортеж із номера рядка та ПОВНИЙ словник (тобто повний рядок, представлений у вигляді словника за стовпчиками). Таким чином, навіть у невідсортованих рядках - виконується пошук за ключовими полями. Розглянемо приклад. Візьмемо два умовні рядки, що йдуть у файлі НЕ послідовно - нехай рядок d1 - має індекс 1, а рядок d2 - індекс 100:

```python
d1={
       "isdn":"+12345678",
       "email":"test123@mailbox.com"
}
d2={
       "isdn":"+12345678",
       "email":"test456@mailbox.com"
}
```

У повному словнику - це буде виглядати так:

```python
{
"isdn":{
"+12345678":
(1,{
       "isdn":"+12345678",
       "email":"test123@mailbox.com"
      })
},
"email":{
"test123@mailbox.com":
      (1,{
       "isdn":"+12345678",
       "email":"test123@mailbox.com"
      })
},
"test456@mailbox.com":
      (100,{
       "isdn":"+12345678",
       "email":"test456@mailbox.com"})}}```
```

Тобто як можна бачити - тому що пошта - не дублюється - її було внесено - як окреме значення - з копією повного словника, тоді як телефон - не було занесено повторно (значення за ключем - не було оновлено), але завдяки тому, що рядок d2 - має такий самий номер телефону - для рядка (словник d2) - за полем телефон - спрацює умова:

```python
if d2[d_k]!="" and d2[d_k] in merged_keys_values_dict[d_k]:
```

Що, своєю чергою, призведе до повного перебору ключів словників і порівнювання їхніх знань - з подальшим об’єднанням неоднакових значень - до першого запам’ятовуваного в загальний словесник рядка (тобто першим словником буде сам рядок d2, а другим словником - раніше запам’ятовуваний словник за номером телефону (рядок d1), який якраз і дублюється для d2):

```python
v=list(merged_keys_values_dict[d_k][d2[d_k]][1].values())
str=data[i+1]
dict_keys=list(merged_keys_values_dict[d_k][d2[d_k]][1].keys())
v2=str.split(base_column_delimetr)
if len(v)==len(v2):
	I=0
	for k1,k2 in zip(v,v2):
		k1 = collapse_line(k1, False)
		k2 = collapse_line(k2, False)
		if k1!=k2:
			original_non_duplicate_dict = merged_keys_values_dict[d_k][d2[d_k]][1]
			if k1 not in k2 and k2 not in k1:
				update_info=additioonal_column_delimetr.join([k1,k2])
				.....
				original_non_duplicate_dict[dict_keys[I]]=update_info
			else:
				if k1 in k2:
					original_non_duplicate_dict[dict_keys[I]] = k2
				elif k2 in k1:
					original_non_duplicate_dict[dict_keys[I]] = k1

data[merged_keys_values_dict[d_k][d2[d_k]][0]]=base_column_delimetr.join(list(original_no
n_duplicate_dict.values()))+"\n"
```

Отримуємо всі значення - першого рядка (словника):

```python
```

```
v=list(merged_keys_values_dict[d_k][d2[d_k]][1].values())
```

Та ключі - даного словника, який запам’ятали раніше:

```
```python
dict_keys=list(merged_keys_values_dict[d_k][d2[d_k]][1].keys())
```
```

Отримуємо поточний рядок, який перевіряємо на можливість дубліката:

```
```python
v2=str.split(base_column_delimetr)
```
```

Звіряємо кількість стовпців:

```
```python
if len(v)==len(v2):
```
```

За кожним стовпчиком - перевіряємо вже не збіг значень - щоб об’єднати різні стовпчики в рядках, що дублюються (наприклад, рядки рівні, окрім стовпчика з електронною поштою)

```
```python
for k1,k2 in zip(v,v2):
              k1 = collapse_line(k1, False)
              k2 = collapse_line(k2, False)
              if k1!=k2:
       ..............
```
```

Об’єднуємо два різних значення:

```
```python
if k1 not in k2 and k2 not in k1:
                            update_info=additioonal_column_delimetr.join([k1,k2])
                            ..................
```
```

І тепер - записуємо оновлений словник (у вигляді рядка через роздільник) - за запам’ятовуваним раніше індексом - у кортежі:

```
```python
data[merged_keys_values_dict[d_k][d2[d_k]][0]]=base_column_delimetr.join(list(original_no
n_duplicate_dict.values()))+"\n"```
```
```

Видалення дублікатів шляхом хешування - перед записом у файл

Пошук дублікатів так само можна розділити на:

– локальний пошук дублікатів - у межах заданого файлу;

– пошук дублікатів під час очищення;

– пошук дублікатів - на етапі постпроцесингу;

– пошук дублікатів щодо зовнішньої БД.

Під час пошуку дублікатів відносно зовнішньої БД - основна проблема полягає в тому, що - відносно самої операції - БД - являє собою чорну шухляду, тобто в нас немає інформації (за відсутності доступу до логів завантаження) про те, які рядки були завантажені в БД.

Розглянемо проблему більш детально. Стоїть завдання виконати пошук значень, що не перетинаються, у двох списках об’єктів словників. Кожен список може містити щонайменше по 100 тисяч об’єктів, і це тільки під час поточної операції довантаження й опрацювання нових значень, так само потрібно враховувати всі попередні завантажені об’єкти.

У цього завдання може бути дві початкові стартові умови.

1. На вхід подається більше одного файлу.
2. На вхід подається один файл.

Розглянемо кожен із випадків. На вхід подається більше одного файлу. У такому випадку, алгоритм складається з таких кроків.

1. Читання пари файлів (докладніше про читання файлів нижче).
2. Якщо довжини списків об’єктів - для двох файлів - не рівні - виконати операцію доповнення меншого списку - порожніми словниками.
3. Провести індексування - створити пару - індекс об’єкта в списку і сам об’єкт.
4. Провести хешування словників - за їхніми значеннями.
5. Якщо присутній файл із попередніми хешами - вшанувати його.
6. Взяти НЕ перетин множин всіх хешів.

7. Знайти індекси об’єктів для решти хешів, що не перетинаються (зворотне індексування).
8. Отримати унікальні словники за знайденими індексами.
9. Оновити файл з усіма хешами - записавши хеші всіх об’єктів словників - з прочитаних файлів.

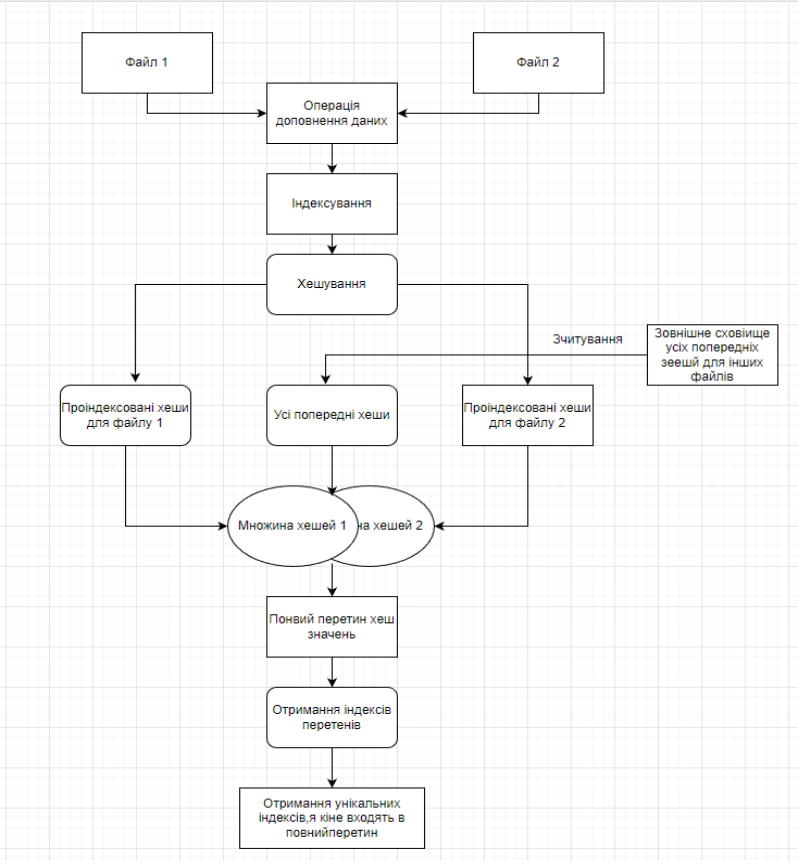

Рисунок 1.4 – Діаграма виконання операції хешування для видалення дублікатів

Для випадку, якщо вхідний файл усього один.

1. Зробити доповнення індексування та хешування всіх словників.
2. Записати отриманий список хешів у файл.
3. Повернути список усіх об’єктів.

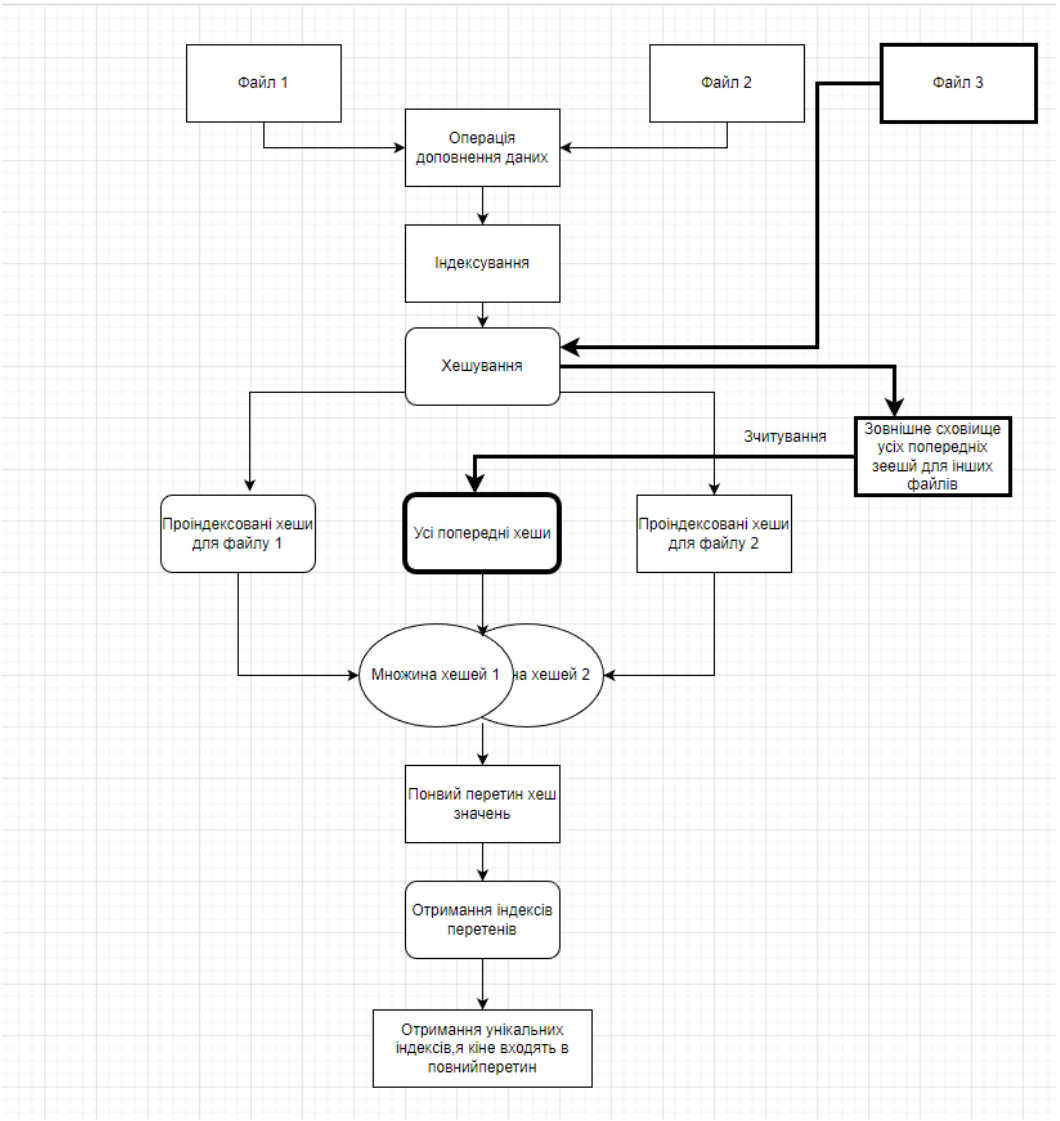

Рисунок 1.5 –Діаграма виконання операції хешування для видалення дублікатів за наявності більше двох вхідних файлів

При читанні списку файлів - циклом - отримуємо ковзне вікно. При читанні парами - може виникати проблема повторного читання вже прочитаних даних - при зміщенні вікна. Для уникнення цієї проблеми - необхідно запам’ятовувати шляхи до вже прочитаних файлів, і якщо в парі, яка потрапила в ковзне вікно, є вже прочитаний файл, то виконувати читання тільки одного непрочитаного файлу, тим самим, зводячи завдання до другого варіанту - пошуку унікальних
значень для одного файлу

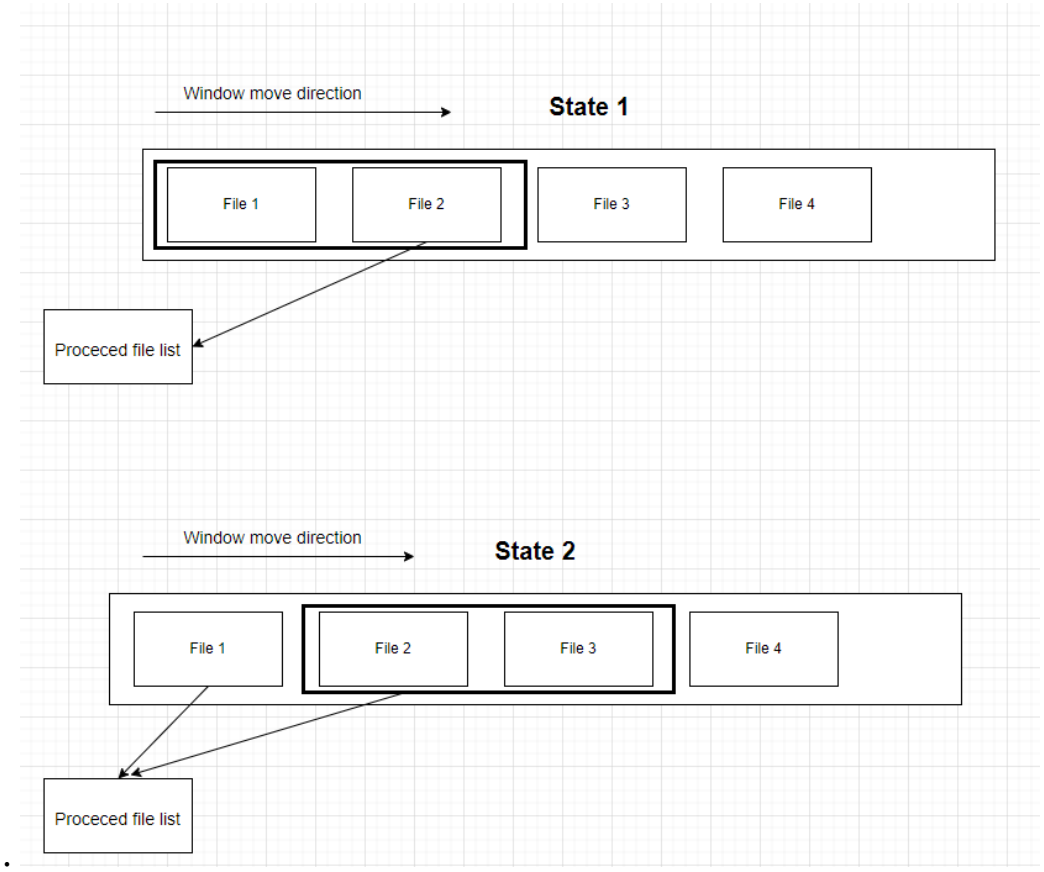

Рисунок 1.6 –Застосування підходу «ковзаюче вікно» при виконанні операції пошуку дублікатів з використанням хешування

Приклад реалізації читання - методом ковзного вікна:

```
```python
if len(files_list)>1:
        for i in range(0,len(files_list)-1):
            if files_list[i] not in proceed_files_list:
                proceed_files_list.append(files_list[i])
                index1=i
            else:
                index1=-1
            if files_list[i+1] not in proceed_files_list:
                proceed_files_list.append(files_list[i+1])
                index2=i+1
            else:
                index2=-1
            loaded_json_data=get_json_objs(files_list,index1,index2)
```
```

Приклад функції читання:

```
```python
def get_json_objs(files_list:list,index1,index2):
    readed_files_data=[]
    inxexes=[]
    if index1!=-1:
```
```

```
        inxexes.append(index1)
    if index2!=-1:
        inxexes.append(index2)
    if len(files_list)>0:
        for l in inxexes:
            try:
                with open(files_list[l],encoding="utf-8") as f:
                    readed_files_data.append(json.load(f))
            except Exception as e:
                print(e)
    return readed_files_data
```

2.3 Маппінг очищених даних

ETL-система - може робити маппінг даних у двох «режимах» роботи.

1. Маппінг даних з фінальною видачею списку стовпців, який потім буде склеєний у рядок.

2. Маппінг даних з фінальною видачею словника стовпців, який потім буде склеєний у рядок.

Кожен варіант має свої переваги та недоліки:

– варіант із використанням списку може призвести до зсуву елементів списку, у разі некоректної маніпуляції з елементом - наприклад, один з елементів фінального списку став порожнім рядком, через що на фінальній стадії перевірки перед конкатенацією - його було відкинуто. Інша проблема може виникнути через змінний розмір списку - по-визначенню - тобто. може виникнути ситуація, наприклад при розмірі списку в п’ять елементів - один з них в результаті обробки був рекурсивно розбитий на дві частини і кожна з них була додана в початковий список з видаленням елемента, що зазнав розбиття - у підсумку, довжина списку стала дорівнювати шести елементам (5-1 +2) - що знову призвело, по-перше, до проблеми зсуву елементів, а по-друге, - до проблеми - "доповнення" решти фінальних списків стовпчиків (рядків) до загальної кількості елементів, наприклад, якщо для будь-яких рядків не виникла ситуація з розподілом значення елементів, наприклад, якщо для будь-яких рядків не виникла ситуація з поділом значення;

– варіант з використанням словника - позбавлений проблем зсуву елементів і проблеми «змінного розміру» рядків, але має один суттєвий недолік - необхідність заздалегідь визначити ключі словника - імена стовпців після маппінгу і його значення для кожного ключа - список початкових стовпців, які необхідно склеїти та обробити як єдине поле (або ж обробити складові частини і склеїти - як результат).

Як можна бачити - варіант з використанням словника є кращим.

Маппінг очищених даних використовує окремий словник:

```python
data_concatenation_dict = {0: [0], 1: [1], 2: [2], 3: [3], 4: [4], 5: [5]}
```
```
def concatenate_data(self, data_concatenation_dict: dict, raw_columns: dict):
    concatenated_columns = dict()
    t = [i for i in raw_columns.keys()]
    t1 = [i for i in raw_columns.values()]
    if len(raw_columns) > len(data_concatenation_dict):
        for k in range(0, len(raw_columns)):
            if k not in r:
                data_concatenation_dict[k] = [k]
    try:
        if len(raw_columns) > 1:
            for k, v in data_concatenation_dict.items():
                for l in v:
                concatenated_columns[t[k]] = " ".join(tmp_l)
        return concatenated_columns
    except Exception as e:
        pass
```

Важливою частиною цієї функції є цей фрагмент:

```python
    t = [i for i in raw_columns.keys()]
    t1 = [i for i in raw_columns.values()]
    if len(raw_columns) > len(data_concatenation_dict):
        for k in range(0, len(raw_columns)):
            if k not in r:
                data_concatenation_dict[k] = [k]
```

Він відповідальний за те, щоб у разі, якщо кількість вихідних очищених стовпців - більша за кількість заданих елементів словника конкатенації - автоматично доповнити словник до потрібної довжини. Наприклад, нехай кількість вихідних очищених стовпців - дорівнюватиме 8, а кількість елементів у словнику - 6:

```python
data_concatenation_dict = {0: [0], 1: [1], 2: [2], 3: [3], 4: [4], 5: [5]}
```

```
```

У такому разі, словник буде доповнено до такого вигляду:

```python
data_concatenation_dict = {0: [0], 1: [1], 2: [2], 3: [3], 4: [4], 5: [5],5: [5],6: [6],7: [7]}
```

При цьому, враховується варіант, коли індекс уже використовується в списку значень словника, наприклад:

```python
data_concatenation_dict = {0: [0], 1: [1,6], 2: [2], 3: [3], 4: [4], 5: [5]}
```

Тоді словник буде доповнено таким чином:

```python
data_concatenation_dict = {0: [0], 1: [1], 2: [2], 3: [3], 4: [4], 5: [5],5: [5],7: [7]}
```

Тобто індекс не буде використовуватися як індекс словника. Оновлена версія функції:

```python
 def concatenate_data(self, data_concatenation_dict: dict, raw_columns: dict,
                          full_keys_list_for_reamer_adding: list):
        concatenated_columns = dict()
        tmp_dict_for_add = {}
        for i in full_keys_list_for_reamer_adding:
            tmp_dict_for_add[i] = ""
        add_dict_to_full([raw_columns, tmp_dict_for_add])
        r = [j for i in data_concatenation_dict.values() for j in i]
        t = [i for i in raw_columns.keys()]
        t1 = [i for i in raw_columns.values()]
        if len(raw_columns) > len(data_concatenation_dict):
            for k in range(0, len(raw_columns)):
                if k not in r:
                    data_concatenation_dict[k] = [k]
        try:
            if len(raw_columns) > 1 and len(raw_columns) == len(data_concatenation_dict):
                for k, v in data_concatenation_dict.items():
                    tmp_l = []
                    for l in v:
                        tmp_l.append(t1[l])
                    concatenated_columns[t[k]] = " ".join(tmp_l)
            return concatenated_columns, tmp_dict_for_add
        except Exception as e:
            print(e)

    def group_data(self, grouping_sign):
        pass
```

В оновленій версії функції враховуються стовпці, які були отримані в результаті операції розгортки - параметр `full_keys_list_for_reamer_adding`. Стовпці, отримані в результаті операції розгортки - додаються до тимчасового словника з порожніми значеннями, після чого відбувається доповнення тимчасового словника - на основі переданого словника з очищеними значеннями:

```
```python
tmp_dict_for_add = {}
for i in full_keys_list_for_reamer_adding:
       tmp_dict_for_add[i] = ""
add_dict_to_full([raw_columns, tmp_dict_for_add])
```
3.Функції очищення ETL-системи
```

Спеціалізовані функції - функції для перевірки та валідації даних, для перевірки даних із певної категорії, наприклад функція перевірки номера мобільного телефону, функція перевірки формату та граничних значень дати народження, функція перевірки адреси.

1. **Мета-функції** - тип функцій, які не належать до спеціалізованих функцій. Узагальнені функції, логіка роботи яких - не залежить від категорії даних. Прикладом такої функції - може слугувати функція `remove_column_by_value` - яка замінює весь стовпчик у csv-файлі, якщо текст у стовпчику починається з певного підрядка.

2. **Функції пошуку за словником** - одні з основних функцій, що застосовуються так само, як допоміжні - для спеціалізованих функцій. Основна ідея цих функцій - така:

– задати початковий словник слів для пошуку, наприклад:

```
```python
             keywords_for_search = {"passport": ["passport", "паспорт"], "job_place":
["организация"], "address": ["адрес"],
                        "itn": ["itn", "инн"], "address_index": ["индекс"],
                        "address_keywords": ["дом", "улица", "корус", "квартира", "ул.",
"кв."],
                        "gender": ["m", "f", "м", "ж", "муж", "жен"]}
               ```
```

Усі значення в цьому словнику задаються в єдиному регістрі, наприклад - нижньому. Для цього словника - створюється - ще один - допоміжний:

```
```python
      tart_keywords_dict_length = {"passport": 2, "job_place": 1, "address": 1, "itn": 2,
"address_index": 1,
                                "address_keywords": 6, "gender": 6}
      ```
```

В якому вказується початкова кількість ключових слів у кожному списку для заданої групи. Як можна бачити - ключі даних словників - збігаються і в другому словнику - вказано кількість елементів у списку першого словника за тим самим ключем.

Потім, у процесі роботи спеціалізованих функцій - вони можуть використовувати словник `keywords_for_search` - для пошуку входжень, старту або закінчення своїх даних - у списку - за ключем цього словника. Основна ідея словника полягає в подальшому його доповненні до всіх можливих комбінацій, наприклад - за допомогою такої функції:

```
      ```python
     def add_dict_to_full_combinations(dict_key: str):
   if len(keywords_for_search[dict_key]) == start_keywords_dict_length[dict_key]:
       if dict_key in keywords_for_search.keys():
           dict_values = keywords_for_search[dict_key].copy()
           for kw in dict_values:
               keywords_for_search[dict_key].append(kw.title())
               keywords_for_search[dict_key].append(kw.upper())
               keywords_for_search[dict_key].append(kw.lower())
   return keywords_for_search
     ```
```

Причому, для економії ресурсів - дана функція буде викликатися, тільки в тому випадку, якщо кількість елементів у списку за заданим ключем - у словнику - дорівнюватиме значенню за цим же ключем у словнику `start_keywords_dict_length` - що свідчить про те, що список ключових слів ще не був доповнений до повного набору комбінацій. Саме для цього і створювався словник - `start_keywords_dict_length` - по-суті, його значення слугують перевіркою - чи потрібно викликати функцію доповнення списку в словнику. Після доповнення

словника, функції пошуку за словником - здійснюють перевірку заданого значення за заданим ключем у словнику.

3. **Функції валідації** – є підмножиною функцій пошуку за словником і, по суті, повністю засновані на їхньому використанні. Єдиною відмінністю, яку можна виокремити для таких функцій, є можливість заміни знайдених значень (входжень) після пошуку за словником - на інше задане значення. Приклад реалізації такої функції:

```python
def check_substring_in_data(data: str, dict_key: str, need_replace: bool):
    if data:
        translated_data = data.strip()
        keywords_dict_search = check_search_dict(dict_key)
        if not keywords_dict_search:
            keywords_dict_search = keywords_for_search
        if keywords_dict_search:
            if need_replace:
                for kw in keywords_dict_search[dict_key]:
                    if kw in data:
                        translated_data = translated_data.replace(kw, " ")
                return translated_data.strip()
            else:
                for kw in keywords_dict_search[dict_key]:
                    if kw in data:
                        return data

    return ""
```

Для другої версії версії функції було додано таку перевірку:

```python
                        if (kw_index + len(kw) < len(translated_data) and (kw_index > 0 and kw_index > len(kw)) and (translated_data[kw_index - 1] == " " and kw_index + len(kw) + 1 < len(translated_data)   and   translated_data[kw_index   +   len(kw)]   ==   "   ")   or can_replace_token_in_middele_without_spaces:
```

Сенс цієї перевірки полягає в такому:

– обмежити заміну токена в рядку - якщо рядок починається з цього токена;

– обмежити заміну токена в рядку - якщо рядок закінчується цим токеном;

– опціонально обмежити заміну токена - якщо в середині рядка і оточений пробілами з двох сторін.

Найпростіше показати її сенс можна на 3 послідовних модифікаціях цієї перевірки - від простішої - до фінальної версії:

Приклад обмеження заміни токена, якщо він знаходиться в кінці рядка:

```python
translated_data="test_test2"
kw="st2"
kw_index=translated_data.index(kw)
print(kw_index)
if kw_index+len(kw)<len(translated_data):
    print("ok")
else:
    print("not ok")
```

Перевірка `if kw_index+len(kw)<len(translated_data):` - гарантує, що за заданим токеном - будуть слідувати символи

Приклад обмеження заміни токена, якщо рядок починається з даного токена або він знаходиться в кінці рядка:

```python
translated_data="2test_test2"
kw="2t"
kw_index=translated_data.index(kw)
print(kw_index)
if kw_index+len(kw)<len(translated_data) and (kw_index>0 and kw_index>len(kw)):
    print("ok")
else:
    print("not ok")
```

У цьому разі важливо звернути увагу на те, що ми не просто перевіряємо індекс входження токена і чи збігається цей індекс з індексом останнього символу рядка, а так само враховуємо і довжину самого токена - що точно свідчить про те, що рядок закінчується на заданий токен: `kw_index + len(kw)<len(translated_data)`.

Наступним поліпшенням перевірки є - перевірка того, що токен входить у рядок НЕ в його крайніх позиціях і оточений пробілами - і тільки за наявності пропусків навколо токена - ми можемо замінити його - `(translated_data[kw_index-1]==" " and kw_index+len(kw)+1<len(translated_data) and translated_data[kw_index+len(kw)]==" ")` - що позбавляє від проблеми випадкової заміни токенів (наприклад, пари символів усередині слова, навіть якщо вони є в словнику заміни)

```python
# translated_data="2test_test2"
# kw="2t"
translated_data="test 2t test"
kw="2t"
kw_index=translated_data.index(kw)
print(kw_index)
if kw_index+len(kw)<len(translated_data) and (kw_index>0 and kw_index>len(kw)) and (translated_data[kw_index-1]==" " and kw_index+len(kw)+1<len(translated_data) and translated_data[kw_index+len(kw)]==" "):
    print("ok")
else:
    print("not ok")
```

І фінальна версія перевірки, яка дає змогу здійснити заміну токена в середині рядка, навіть якщо навколо токена немає пробілів - що робить цю перевірку опціональною:

```python
# translated_data="2test_test2"
# kw="2t"

translated_data="test 2ttest"
kw="2t"
can_replace_token_in_middele_without_spaces=True
kw_index=translated_data.index(kw)
print(kw_index)
if (kw_index+len(kw)<len(translated_data) and (kw_index>0 and kw_index>len(kw)) and (translated_data[kw_index-1]==" " and kw_index+len(kw)+1<len(translated_data) and translated_data[kw_index+len(kw)]==" ")) or can_replace_token_in_middele_without_spaces:
    print("ok")
else:
    print("not ok")
```

Так само, додано параметр - `need_replace_full` - що дає змогу повністю видалити рядок (зробити його порожнім) - у разі знаходження в ньому ключового слова.

4. **Функції трансформації** - по-суті - ті самі функції валідації, але винесені в окрему групу, тому що, на відміну від перших, можу застосовувати більш-складні (будь-які) трансформації до даних, а не тільки просту заміну знайдених входжень.

Приклад використання ієрархічності функцій:

```python
def validate_passport(data: str):
    return check_substring_in_data(data, "passport", True)
```

– `validate_passport` - спеціалізована функція;

– `check_substring_in_data` - функція валідації, що викликає функцію пошуку за словником (доповнює словник і отримує посилання на його значення - за ключем - повний список ключових слів для пошуку - шляхом виклику функції `check_search_dict`) - у процесі своєї роботи - для пошуку однієї з варіацій слова "паспорт" у вхідних даних, а параметр - `True` - вказує на необхідність заміни даного входження (на початку рядка) - на порожній рядок (видалення входження на початку рядка).

Приклад оновленої версії функції - містить у собі - можливість додавання варіантів транслітерації елементів вихідного списку:

```python
def add_dict_to_full_combinations(dict_key: str,need_transliterate_dict_items=False):
    if len(keywords_for_search[dict_key]) == start_keywords_dict_length[dict_key]:
        if dict_key in keywords_for_search.keys():
            dict_values = keywords_for_search[dict_key].copy()
            dict_values = check_delimeters_in_start_list(dict_values)
            for kw in dict_values:
                keywords_for_search[dict_key].append(kw.title())
                keywords_for_search[dict_key].append(kw.upper())
                keywords_for_search[dict_key].append(kw.lower())
                if need_transliterate_dict_items:
                    if not kw.isnumeric() and kw!=" " and kw!="-" and kw!="_":
                        try:
                            keywords_for_search[dict_key].append(translit(kw.title(),
'ru'))
                            keywords_for_search[dict_key].append(translit(kw.upper(),
'ru'))
                            keywords_for_search[dict_key].append(translit(kw.lower(),
'ru'))



                            keywords_for_search[dict_key].append(translit(kw.title(),
reversed=True))
                            keywords_for_search[dict_key].append(translit(kw.upper(),
reversed=True))
                            keywords_for_search[dict_key].append(translit(kw.lower(),
reversed=True))
                        except Exception as e:
                            pass
                keywords_for_search[dict_key]=list(set(keywords_for_search[dict_key]))
    return keywords_for_search
```

Говорячи про питання доповнення словника комбінацій токенів - важливо ввести таке поняття, як класифікація символів на:

– символи класу роздільник;

– решта символів.

Символи класу роздільник - це символи, які присутні між усіма іншими символами. Дані символи не слід плутати зі спец символами - такими як escape-послідовності.

Особливістю символу класу роздільник - є те, що він може бути представлений у кількох варіаціях, що ускладнює завдання його заміни або екранування - серед решти символів, якщо це необхідно.

Прикладами символу класу роздільник - може бути - апостроф, тому що навіть у стандартах ASCII і Unicode- дані символи можуть бути представлені в декількох варіаціях (https://www.cl.cam.ac.uk/~mgk25/ucs/quotes.html) - наприклад:

```
```
apostrophes_types=["`","`","`"]
```
```

Саме з метою визначення символу як символу класу роздільник і додавання інших його подань - у підмножину для подальшої генерації комбінацій слів - було додано таку функцію:

```
```python
def
check_delimeters_in_start_list(start_list:list,additional_delimetr_chars:
dict={}):
    checked_words=set()
    for w in start_list:
        if w not in checked_words:
            if "`" in w or "`" in w or "`" in w:
                apostrophe=""
                if "`" in w:
                    apostrophe="`"
                if "`" in w:
                    apostrophe="`"
                if "`" in w:
                    apostrophe="`"

                apostrophes_types=["`","`","`"]
                # apostrophes_types.remove(apostrophe)

                for additional_apostrophe in apostrophes_types:
                    new_word=w.replace(apostrophe, additional_apostrophe)
```
```

```
                    start_list.append(new_word)
                    checked_words.add(new_word)
            for k,v in additional_delimetr_chars.items():
                if k in w:
                    for new_del in v:
                        new_word=w.replace(k,new_del)
                        start_list.append(new_word)
                        checked_words.add(new_word)
            checked_words.add(w)
    updated_list= list(set(start_list))
    updated_list.sort()
    return updated_list
```

Поняття символу класу роздільник - так само породжує ще одну проблему, крім його множинних уявлень - деякі ЯП, наприклад python, неправильно використовують стандартні функції зміни регістру, якщо в рядку присутні символи цього типу:

Рисунок 2.1 –Приклад неправильної роботи зі спец-символами -стандартними засобами мови Python

Саме тому, такі випадки доводиться обробляти окремо:

```
```python
if "`" in kw or "`" in kw or "`" in kw:
             word_title=kw[0].upper()+kw[1:]
```
```python
def add_dict_to_full_combinations(dict_key: str, need_transliterate_dict_items=False):
    if len(keywords_for_search[dict_key]) == start_keywords_dict_length[dict_key]:
        if dict_key in keywords_for_search.keys():
            dict_values = keywords_for_search[dict_key].copy()
            dict_values = check_delimeters_in_start_list(dict_values)

            for kw in dict_values:
                keywords_for_search[dict_key].append(kw.title())
                keywords_for_search[dict_key].append(kw.upper())
                keywords_for_search[dict_key].append(kw.lower())
                if need_transliterate_dict_items:
                    if not kw.isnumeric() and kw != " " and kw != "-" and kw != "_":
                        try:
                            ## for test
                            # if kw=="ім'я":
                            #     print("d")

                            if "`" in kw or "`" in kw or "`" in kw:
                                word_title=kw[0].upper()+kw[1:]
                                keywords_for_search[dict_key].append(word_title)
                                keywords_for_search[dict_key].append(translit(word_title,
'ru'))
                                keywords_for_search[dict_key].append(translit(word_title,
reversed=True))

                                #
keywords_for_search[dict_key].append(string.capwords(kw))
                                #
keywords_for_search[dict_key].append(translit(string.capwords(kw), 'ru'))
                                #
keywords_for_search[dict_key].append(translit(string.capwords(kw), reversed=True))
                            else:
                                keywords_for_search[dict_key].append(kw.title())

keywords_for_search[dict_key].append(translit(string.capwords(kw.title()), 'ru'))

keywords_for_search[dict_key].append(translit(string.capwords(kw.title()),
reversed=True))

                            keywords_for_search[dict_key].append(kw.upper())
                            keywords_for_search[dict_key].append(translit(kw.lower(),
'ru'))
                            keywords_for_search[dict_key].append(translit(kw.upper(),
'ru'))
                            keywords_for_search[dict_key].append(translit(kw.lower(),
reversed=True))
                            keywords_for_search[dict_key].append(translit(kw.upper(),
reversed=True))
```
```

```
                    except Exception as e:
                        pass
            keywords_for_search[dict_key] = list(set(keywords_for_search[dict_key]))
```

Висновки до розділу 2

Як було продемонстровано у даному розділі – структуризація даних перед їх обробкою – є одним з ключових етапів, який впливає на правильність кінцевого результату. Структуризація даних досягається шляхом виконання операції «маппінгу» (англ. Mapping). Сам маппінг може відбуватися за індексами вже очищених даних або ж з використанням заздалегідь визначеного словника з заданим набором ключів, що дозволяє не піклуватися про послідовність зберігання значень та їх можливий зсув.

Для поліпшення процесу обробки даних, а саме – для придання гнучкості конвеєру даних – було продемонстровано застосування принципів ООП з метою створення незалежних об'єктів, що можуть інкапсулювати в собі логіку обробки даних з певного інформаційного домену, враховуючи специфіку даних.

Також було розглянуто принцип роботи з роздільниками даних, та продемонстровано, що дані можуть бути розділені за основним та додатковим роздільником, що впливає на логіку їх подальшої обробки.

Наприкінці розділу було використано наведені приклади маппінгу даних – для отримання фінального масиву вже очищених та структурованих даних.

3 ІНСТРУМЕНТАЛЬНІ ЗАСОБИ ІНТЕЛЕКТУАЛЬНОЇ СИСТЕМИ ОБРОБКИ ВЕЛИКИХ ДАНИХ

3.1 Очищення та трансформація даних

Наведемо результат очищення та трансформації даних на прикладі роботи з номером мобільного телефону.

Початковий стан даних, які можуть вважатися номером мобільного телефону, можна розділити на такі групи.

1. За типом підготовленості даних:

– у рядку міститься тільки номер телефону;

– у рядку міститься номер телефону та інші символи або значення.

2. За типом мобільного телефону:

– телефон у повному - міжнародному форматі;

– телефон не в повному форматі:

– відсутній знак `+`;

– відсутній код країни.

Якщо дані недостатньо підготовлені - тобто крім самого номера - в рядку присутні й інші дані, але телефон міститься в повному форматі (нехай і без символу `+`) - досить використовувати регулярні вирази.

В іншому випадку - необхідно розробити систему правил для доповнення і валідації номера мобільного телефону.

Отже, вимоги та кроки, які застосовуються до даних, якщо вони належать до цієї категорії.

1. Видалення всіх не числових символів (прибираємо з рядка все, крім цифр).

2. Перевірка та доповнення номера мобільного телефону - ґрунтується на системі правил, яку можна представити в такому вигляді:

```
```python
```
```

```
number_matching_rules ={
"ua": [("380", (12, ukrOperatorsCodes)),
        ("38", (12, ukrOperatorsCodes1)),
        ("3", (12, ukrOperatorsCodes2))],
 "ru": [("7", (11, RussianOperatorCodes)),
        ("7", (11, dnr_operator_codes)),
        ("7", (11, lnr_operator_codes)),
        ("8", (11, RussianOperatorCodes))],
        .....
        }
```

Розглянемо наведену систему правил. Як можна бачити - це словник, де ключ - це код країни, а значення - список кортежів. Розглянемо приклад кортежу.

1. Першим елементом кортежа є префікс коду країни, причому в даному випадку, якщо довжина префікса більша за один символ, зазначено всі можливі варіанти префікса, але надалі такий запис можна замінити на автоматичну генерацію всіх можливих варіантів і перебір за циклом;

2. Другим елементом кортежу - так само є кортеж, який складається з:

– цілочисельного значення - повної довжини валідного номера з кодом країни (без урахування символу `+`);

– посилання на допоміжний список, у якому містяться коди телефонних операторів для даної країни, наприклад.

```python
      ukrOperatorsCodes=["67",    "68",    "96",    "97",    "98","50",    "66",    "95",
"99","63","93","91","73"
                                                ,"39","92","94"]

ukrOperatorsCodes1=["067",    "068",    "096",    "097",    "098","050",    "066",    "095",
"099","063","093","091","073"
                                                  ,"039","092","094"]
ukrOperatorsCodes2=["8067",  "8068",  "8096",  "8097",  "8098","8050",  "8066",  "8095",
"8099","8063","8093","8091","8073"
                                                ,"8039","8092","8094"]

RussianOperatorCodes=["902","950","952","900","903","905","906","909","951","960","961","
962","963","964","965","968","969","980","983","986","955"

,"997","970","971","995","958","985","933","996",
"999","954","941","904","908","953","977","930","939","966","981","984","901","989","920"
,"921","922","923","925","926","927"
,"928","929","937","910","911","912","913","914","915","916","917","918","919","978","982
","983","985","986","987","988"
,"912","923","924","931","934","936","967","968","982","983","985","986","988","991","992
","993","994","932","938"]
```

Слід звернути увагу на наступне:

– наприклад, для коду країни України - можливі три варіанти його запису щодо вхідних даних:

– 380 - якщо вхідний рядок починається одразу з коду оператора;

– 38 - якщо вхідний рядок починається із символу `0` та коду оператора;

– 3 - якщо вхідний рядок починається із символів `80` і коду оператора;

– відповідно, коди мобільних операторів - так само продубльовані у вигляді трьох списків для цих випадків:

– якщо вхідний рядок починається відразу з коду оператора - використовується список `ukrOperatorsCodes` і потім, до вихідних даних - додається код країни у форматі `380`;

– якщо вхідний рядок починається із символу `0` і коду оператора - використовується список `ukrOperatorsCodes1` і потім, до вихідних даних - додається код країни у форматі `38`;

– якщо вхідний рядок починається з символів `80` і коду оператора - використовується список `ukrOperatorsCodes2` і потім, до вихідних даних - додається код країни у форматі `3`.

Алгоритм перевірки та доповнення номера - до повного – подано нижче.

1. Видалення всіх не числових символів (прибираємо з рядка все, крім цифр).
2. Отримання кортежу за ключем - кодом країни для перевірки номера.
3. Перевірка - чи починається вхідний рядок із зазначеного в кортежі - коду країни:

– так - виконується перевірка, чи відповідає довжина вхідного рядка - заданому цілочисельному значенню - довжині повного номера:

– так - до рядка додається символ `+` (опціонально) і він вважається валідним;

– ні - рядок вважається не валідним;

– ні - відбувається перевірка, чи починається рядок з одного з кодів оператора в заданому допоміжному списку:

– так - рядок доповнюється кодом країни, після чого - перевіряється його довжина - він має відповідати довжині номера в міжнародному форматі - для цієї країни. Якщо операція перевірки на довжину - успішна - до рядка додається символ `+` (опціонально) і він вважається валідним;

– ні - рядок вважається не валідним.

3.2 Особливості перевірки текстових даних за словниками

```python
def translate_name(data: str):
    if data:
        stripped_data = data.strip()
        stripped_data = " ".join(stripped_data.split())
        emoji_less_text = demoji.replace(stripped_data, "")
        cyrillic_letters = u"абвгдеёжзийклмнопрстуфхцчшщъыьэюяАБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯІіЄєЇї"
        allowed_chars = cyrillic_letters + ascii_letters + whitespace
        additional_allowed_charters="".join([i for i in list(units_params["name"]["additional_skip_bad_charters"])])
        allowed_chars = cyrillic_letters + ascii_letters +whitespace+additional_allowed_charters
        translated_data = "".join([c for c in emoji_less_text if c in allowed_chars])
        if units_params["name"]["need_transliterate_name"]:
            translated_data = transliterate_data(translated_data)
        return translated_data.title()
```

Зверніть увагу, на рядок:

```python
additional_allowed_charters="".join([i for i in list(units_params["name"]["additional_skip_bad_charters"])])
```

Додає до списку дозволених символів - елементи множини, яку задають як параметр юніта (див. параметризацію юніта):

```python
"additional_skip_bad_charters":set()
```

Приклад параметризації:

```python
units["about"].set_unit_params({"additional_skip_bad_charters":
set(additioonal_column_delimetr)})
```

Для перевірки імен, в `keywords_for_search` - було додано ключ `bad_names` і `bad_names_by_places`:

```
```python
keywords_for_search = {"passport": ["passport", "паспорт"],
                       "job_place": ["организация", "должность", "position", "нет", "не
указано"], "address": ["адрес"],
                       "itn": ["itn", "инн"], "address_index": ["индекс"],
                       "address_keywords": ["дом", "улица", "корус", "квартира", "ул.",
"кв."],
                       "gender": ["m", "м", "муж", "ч", "чол", "man", "1", "ж", "f", "жен",
"жін", "0", "woman"],
                       "bad_names": ["1", "2", "3", "4", "5", "6", "7", "8", "9", "0",
"test", "qwerty", "user", "name",
                                     "имя",    "ім'я",    "імя",    "тест",    "тестовый",
"пользователь", "delete", "deleted",
                                     "нет","no","немає","не              встановлено","не
установлено","не  указано","не  указанно","не  вказано","не  вказанно","не  вказане","не
вказанне","Імя","цена","ціна","хороший","хорошая",

"хозяин","хазяин","хозяїн","хазяїн","хозяін","хазяін","господар","владелец","owner",r"\n"
,"defis","сервис","отдел","администрация","service","student","студент","doc","he","she",
"it","он","она","оно","neznakomka","neznakomec","незнакомка","незнкаомец"],

"garbage_values":["null","none","empty","неизвестно","невідомо","не
установлено","неустановлено","невстановлено","не встановлено",
                                     "невстановленно","не
встановленно","неизвестен","не известен","не установлена","неустановлена",
                                     "не
установленна","неустановленна","\\n","dark","fix","dream","ghost","hell","gold"],
                       "bad_names_by_places":["магазин",    "храм",    "церковь","салон",
"торговый центр", "сто", "маникюр", "отдел", "кафе", "ресторан", "аптека", "гостиница",
"отель",   "мотель",   "театр",   "кинотеатр",   "закусочная",   "заправка",   "музей",
"автосалон","компания","фирма","бюро"],

"garbage_values_with_eng_symb":["НЕВСТАНОВЛЕНО","НЕВІДОМО","НЕИЗВЕСТЕН"],
                       "birthdate":[r"\n"],
                       "countries_list":["албания",    "алжир",    "андорра",    "ангола",
"ангилья", "антигуа и барбуда", "аргентина", "армения", "аруба", "австралия", "австрия",
"азербайджан",  "багамы",  "бахрейн",  "бангладеш",  "барбадос",  "беларусь",  "бельгия",
"белиз", "бенин", "бермуды", "боливия", "бонайре, синт-эстатиус и саба", "босния и
герцеговина", "ботсвана", "бразилия", "британская территория в индийском океане", "бруней-
даруссалам", "болгария", "буркина-фасо", "бурунди", "камбоджа", "камерун", "канада", "кабо-
верде", "карибские нидерланды", "каймановы острова", "центральноафриканская республика",
"чад", "чили", "китай", "остров Рождества", "кокосовые (килинг) острова", "колумбия", "
 …… ]

'html_tag_bad_charter_list':list(html_tag_bad_charter_list[0].values())[0],
                                  "animals_list":["кошка", "cat", "собака", "dog", "птица",
"bird",  "рыба",  "fish",  "корова",  "cow",  "лошадь",  "horse",  "свинья",  "pig",  "овца",
"sheep", "заяц", "rabbit", "лев", "lion",
                                                  "тигр",   "tiger",   "слон",   "elephant",
"крокодил", "crocodile", "обезьяна", "monkey", "жираф", "giraffe", "кенгуру", "kangaroo",
"лиса", "fox", "волк", "wolf", "медведь", "bear",
                                                  "змея",    "snake",    "еж",    "hedgehog",
"барсук", "badger", "бурундук", "chipmunk", "белка",
                                                   "squirrel",  "морская  свинка",  "guinea
pig",                    "хорек",                    "hamster",                    "скунс",
"skunk","жаба","лягушка","frog","птица","bird","орел","eagle","ёж","ёжик","ежик","hedgeho
g"],
                       "seasons":["весна",     "весна",     "spring","лето",     "літо",
"summer","осень", "осінь", "autumn","зима", "зима", "winter",
```
```

```
                                    "ясно",     "ясно",     "clear","облачно",     "хмарно",
"cloudy","дождь", "дощ", "rain","снег", "сніг", "snow","гроза", "гроза",
                                    "thunderstorm","ветренно", "вітряно", "windy","туман",
"туман", "foggy","жарко", "жарко", "hot","холодно", "холодно",
                                    "cold","пасмурно",  "похмуро",  "overcast","ненастье",
"ненастья", "stormy","прохладно", "прохолодно", "cool","wind"],
                        "states_of_aggregation":["газ", "газ", "gas","жидкость", "рідина",
"liquid","твердое", "тверде",
                                                 "solid","плазма",              "плазма",
"plasma","кристалл", "кристал", "crystal"

,"жидкость","рідина","вода","liquid","water","vapor","лед","лід","ice"] }
```

І відповідне значення за таким самим ключем - для словника - `start_keywords_dict_length`:

```
```python
tart_keywords_dict_length = {"passport": 2, "job_place": 5, "address": 1, "itn": 2,
"address_index": 1,
                                "address_keywords":   6,   "gender":   13,   "bad_names":
66,"garbage_values":28,"garbage_values_with_eng_symb":3,"birthdate":1,"bad_names_by_place
s":23,

"countries_list":197,"countries_list_eng":197,"countries_list_translit":197,"html_tag_bad
_charter_list":173,"animals_list":71,"seasons":55,"states_of_aggregation":27}
```
```

`bad_names` і `bad_names_by_places` - було рознесено на два окремі словники, хоча обидва використовуються для перевірки імені - з метою оптимізації, тому що словник `bad_names` - використовується для побудови ВСІХ можливих комбінацій, в той час, як значення словника - `bad_names_by_places` - повинні використовуватися для швидкої перевірки, без сполучень значень - з метою відсіяти, наприклад, такі варіанти:

– «Перукарня Олена»;

– церква святого Олега.

Тобто коли рядок містить ім’я, яке навіть може пройти валідацію за словником імен, але формально цей стовпчик заданого рядка - не буде іменем власним.

Так само, для валідної перевірки імен - необхідно так само вводити поняття алфавітів для різних локалей:

```
```python
ru_alphabet_symbols=u"абвгдеёжзийклмнопрстуфхцчшщъыьэюяАБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ"
```
```

```
ua_alphabet_symbols=u"абвгдєжзийклмнопрстуфхцчшщъыьєюяАБВГДЄЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЄЮЯІіЇї"
en_alphabet_symbols=ascii_letters

class str_locales(enum.Enum):
    ru="ru"
    ua="ua"
    en="en"
```

І поняття граматичного аналізу:

```python
class grammar_lexem_type:
    include_token="include"
    not_include_token="exclude"
    include_but_need_be_part_of_large_token="include_but_need_be_part_of_large_token"
    not_include_token_with_inner_part = "exclude_with_inner_part"

word_grammar_rules = {
    ("ru", "ua"): {"йц": grammar_lexem_type.include_but_need_be_part_of_large_token, "дк": grammar_lexem_type.not_include_token,
                   "авп":           grammar_lexem_type.not_include_token,           "кфх": grammar_lexem_type.not_include_token,
                   "ава":           grammar_lexem_type.not_include_token,           "ыав": grammar_lexem_type.not_include_token,

"ду":grammar_lexem_type.include_but_need_be_part_of_large_token,"св":grammar_lexem_type.include_but_need_be_part_of_large_token,

"фы":grammar_lexem_type.not_include_token,"ыв":grammar_lexem_type.not_include_token,"ыф": grammar_lexem_type.not_include_token,
                   "вап":grammar_lexem_type.not_include_token},
                   "вап":grammar_lexem_type.not_include_token,
                   "вад": grammar_lexem_type.include_but_need_be_part_of_large_token,
                   "вал": grammar_lexem_type.include_but_need_be_part_of_large_token,
                   "каа": grammar_lexem_type.include_but_need_be_part_of_large_token,
                   "днв": grammar_lexem_type.not_include_token,
                    "евб": grammar_lexem_type.not_include_token,
                    "згм": grammar_lexem_type.not_include_token,
                    "ксв": grammar_lexem_type.not_include_token,
                    "ктск": grammar_lexem_type.not_include_token,
                    "мас": grammar_lexem_type.not_include_token,
                    "пнрк": grammar_lexem_type.not_include_token
                   },

"en"):{"fg":grammar_lexem_type.not_include_token,"df":grammar_lexem_type.not_include_token,"dfdg":grammar_lexem_type.not_include_token,

"fktr":grammar_lexem_type.not_include_token,"trctq":grammar_lexem_type.not_include_token,
"dsfds":grammar_lexem_type.not_include_token,

"dsfdsfdsf":grammar_lexem_type.not_include_token,"gfhfd":grammar_lexem_type.not_include_token,
            "grrh":grammar_lexem_type.not_include_token,

"fhd":grammar_lexem_type.not_include_token,"mpgt":grammar_lexem_type.not_include_token,
            "qtk":grammar_lexem_type.not_include_token,
            "gr*lf":grammar_lexem_type.not_include_token_with_inner_part,
```

```
            "wp":grammar_lexem_type.not_include_token,
             "ws":grammar_lexem_type.not_include_token,
             "wq":grammar_lexem_type.not_include_token,
             "ww":grammar_lexem_type.not_include_token,
             "asd":grammar_lexem_type.not_include_token,
             "xx": grammar_lexem_type.not_include_token,
             "ads": grammar_lexem_type.not_include_token,
             "yfm": grammar_lexem_type.not_include_token,
             "qsz": grammar_lexem_type.not_include_token,
             "wjf": grammar_lexem_type.not_include_token,
             "yuw": grammar_lexem_type.not_include_token,
             "qwr": grammar_lexem_type.not_include_token,
             "qwe": grammar_lexem_type.include_but_need_be_part_of_large_token,
             "qw": grammar_lexem_type.include_but_need_be_part_of_large_token,
             "asf": grammar_lexem_type.not_include_token_with_inner_part,
             "asc": grammar_lexem_type.include_but_need_be_part_of_large_token
             }}
```

Так наприклад токени:

```
```python
                                          "вад":
grammar_lexem_type.include_but_need_be_part_of_large_token,
                   "вал": grammar_lexem_type.include_but_need_be_part_of_large_token,
                   "каа": grammar_lexem_type.include_but_need_be_part_of_large_token,
```
```

Можуть зустрічатися як частини імен, наприклад: `Вадим,Валерій,Валерія,Каафа`. Окремим важливим типом лексеми для граматичного аналізу - є - тип - `include_but_need_be_part_of_large_token` - який дозволяє входження токена в рядок - аналогічно до лексеми `include_token`, але робить обов`язковою умову, що б за даним токеном слідували інші символи, що дає змогу, наприклад, здійснити валідацію імені `Дуров`, але відкинути невалідне ім`я із двох символів: `Ду`.

Своєю чергою тип токена - `not_include_token_with_inner_part` - перевіряє входження всіх підтокенів, отриманих за розбиттям граматичного токена за символом зірочки`gr*lf` - наприклад цей токен не пропустить сміттєве значення - `Grolf`, при тому, що:

Так, в англійській мові можна зустріти імена, в яких входить підрядок "Gro" або "gro". Це може бути як власні імена, так і прізвища. Наприклад:

```
Імена:
>1. Gregory
>2. Grover
>3. Grogan
```

```
>
Прізвища:
>1. Grove
>2. Groves
>3. McGregor
```

Так, наприклад, токен - виду:

```
```python
"asf": grammar_lexem_type.not_include_token_with_inner_part,
```
```

Із зазначенням множинної кількості підстановних символів - `` - дають змогу відсіювати наступний тип рядків:

```
```
Asf Sza
```
```

Включно з пробілами та наступними символами - таким чином, роблячи `asf` - основним «підтокеном» - і використовуючи множинну підстановку символів - ми маємо змогу виключити будь-яку можливу кількість варіантів, що відповідатиме даному патерну, наприклад:

```
Asf Sza
Asf<пробел>
Asfd
```

Використовується чіткий поділ на підмножини алфавітів:

```python
        cyrillic_letters=ru_alphabet_symbols+ua_alphabet_symbols
        cyrillic_letters="".join(frozenset(list(cyrillic_letters)))
        allowed_chars = (cyrillic_letters + en_alphabet_symbols
+whitespace+additional_allowed_charters)
```

Де змінна `male_name` - список російських і українських чоловічих імен і так само, додано виклики двох додаткових функцій - `check_grammar_rules` - для перевірки за допомогою правил для входження токенів (див. нижче) і `check_initials_in_name` - для перевірки окремого випадку, коли ПІБ - представлене у вигляді прізвища та ініціалів (див. нижче):

```python
translated_data=check_grammar_rules(translated_data)
translated_data=check_initials_in_name(translated_data)
```

Важливо зазначити, що обмеження для довжини імені - встановлено саме в один, а не в два і більше символи:

```python
if len(data)<=1:
      return ""
```

Оскільки існують валідні імена, що складаються з двох символів, наприклад – «Ян». Зверніть увагу на додаткову перевірку, яка дає змогу вмикати або вимикати нормалізацію алфавіту залежно від параметризації юніта:

```python
      if ("need_normalize_alpahabet" in units_params["name"].keys() and
units_params["name"]["need_normalize_alpahabet"]) or ("need_normalize_alpahabet" not in
units_params["name"].keys()):
            data = replace_char_by_hex_dict(data,
car_number_symb_replace,char_repl_dir=char_replace_direction.from_en_to_cyrilic)
```

Так само, для перевірки імені - розумно використовувати функцію для «колапсу» рядка: (див. опис операції "колапс рядка")

```python
translated_data=collapse_line(translated_data,False)
```

Для уникнення таких випадків, як:

```
Иванов Иван Иванов Иван Иван
```

Перевірка - на послідовну повторюваність символів:

```python
# check symbols repeats
        max_symbol_repeat=3
        for ch in list(cyrillic_letters+en_alphabet_symbols):
            if ch*max_symbol_repeat in stripped_data:
                return ""
```

Дозволяє уникнути рядків - виду:

```
aaaaabcd
Teeeeest test
Some Naaaaaaame
```

Дуже важливою частиною логіки перевірки імен - є **операція нормалізації алфавіту** - яка передбачає приведення всіх символів рядка - до єдиного алфавіту:

```python
replace_char_by_hex_dict(data,car_number_symb_replace,char_repl_dir=char_replace_directio
n.from_en_to_cyrilic)
```

Приклад списку юнітів, які підлягають нормалізації алфавіту:

– email;

– ім’я;

– address.

Перед процедурою нормалізації алфавіту має передувати приведення рядків до єдиного регістру, причому до того регістру, символи якого є ключами у словнику нормалізації:

Приклад видалення дубліката пошти, шляхом приведення до єдиного регістру і нормалізації алфавіту.

Рисунок 2.2 – Приклад видалення дублікату адреса електронної пошти з використанням нормалізації алфавіту

У цьому прикладі, пошта - `Tkachenko.irina.63@gmail.com` - спочатку мала латинську букву `T` - що не робило її ідентичною з поштою `tkachenko.irina.63@gmail.com` - таким чином виникало дублювання:

```
Tkachenko.irina.63@gmail.com|tkachenko.irina.63@gmail.com
```

Причому, якщо не переводити пошту `Tkachenko.irina.63@gmail.com` у режимі нормалізації алфавіту `char_repl_dir=char_replace_direction.from_cyrilic_to_en` - після заміни за словником:

`Tkachenko.irina.63@gmail.com` - вже матиме англійську `T`, але так само не збігатиметься з `tkachenko.irina.63@gmail.com` - що все одно призведе до появи дублікатів.

Так само, розумно токенезувати частини імені та перевірити їхню довжину:

```python
            if " " in translated_data:
                translated_data_parts=translated_data.split(" ") # Н М ; # Таня Р
                translated_data=" ".join([p for p in translated_data_parts if len(p)>1])
                translated_data=translated_data.strip()
```

```
```

Так само при такій перевірці, потрібно враховувати, що довжина для кожного з токенів - не обов'язково повинна бути однакова. Наприклад рядок: `Ira Ed Cb` - містить коректне ім'я `Ira` і сміттєві токени, для яких піднімається межа мінімальної довжини:

```python
#check tokens length:
                if              ("use_hard_delete_mode_for_tokenized_name"              in
units_params["name"].keys()                                                             and
units_params["name"]["use_hard_delete_mode_for_tokenized_name"]):
                                use_hard_delete_mode_for_tokenized_name=True
                    else:
                        use_hard_delete_mode_for_tokenized_name=False
            if " " in translated_data:
                tmp_parts_list=[]
                translated_data_parts = translated_data.split(" ")
                for p in range(0,len(translated_data_parts)):
                    min_part_l=2
                    if "." not in translated_data_parts[p] and len(translated_data_parts)>1
and p>0:
                        min_part_l=3
                    if            len(translated_data_parts[p])>=min_part_l            and
len(translated_data_parts[p])<=15:
                        tmp_parts_list.append(translated_data_parts[p])
                if use_hard_delete_mode_for_tokenized_name:
                    if len(tmp_parts_list)<len(translated_data_parts):
                        translated_data=""
                else:
                    translated_data=" ".join(tmp_parts_list).strip()```
```

При цьому, для першого токена розумно залишити нижню межу - дорівнює двом - прикладом чого може служити рядок: `Ян а бв г` - з коректним ім`ям - `Ян`.

При цьому, за мінімальної довжини імені, що дорівнює двом - слід додати таку перевірку:

```python
        if len(stripped_data.lower())==2:
            if stripped_data[0].lower()==stripped_data[1].lower():
                return ""
```

Булева змінна `use_hard_delete_mode_for_tokenized_name` - вказує на те, чи потрібно робити рядок порожнім, якщо з початкового списку токенів - після підняття мінімального кордону довжини токена - було видалено якісь токени - тобто якщо довжина початкового списку токенів і списку токенів - для всіх, окрім

першого, із більшою грацією довжини - розрізняються, ім’я вважають невалідним, його має бути видалено.

Висновки до розділу 3

У третьому розділі кваліфікаційної роботи магістра – було продемонстровано використання граматичних правил відповідно до токінезовнаих представлень даних, що дозволяє вилучати та виокремлювати такі типи даних як власні імена та назви об’єктів.

Окремо було продемонстровано техніки доповнення даних за певеною пред-визначеною структурою на підставі непрямих ознак або тільки частні доступних валідних даних. Для демонстрації запропонованих ідей – було використано приклади з залученням номерів мобільних телефонів, специфічність яких дозволяє обробляти різні випадки їх використання, а саме – виконувати визначення коду країни та відповідного мобільного оператора – за певним списком правил – з метою отримати номер мобільного телефону у міжнародному форматі у якості кінцевого результату.

ВИСНОВКИ

У даній магістерській кваліфікаційній роботі було продемонстровано розробку автономної напів-автоматизованої системи обробки великих даних з використанням конфігураційних файлів у форматі JSON – які були використані для швидкого налаштування системи та дозволили виконувати валідацію та очищення даних у різних інформаційних доменах - тобто даних, що відносяться до різної предметної галузі. Гнучкість обробки даних великого об’єму була достянена завдяки можливості розбиття вхідного файлу або потоку на окремі частини, очищенні дані з яких – об’єднуються наприкінці процесу роботи системи.

Отримана архітектура застосунку дозволяє не тільки швидко виконувати налашьування системи,але й без особливих проблем додавати новий функціонал до застосунку – без внесення змін у логіку основного конвеєра даних – шляхом створення незалежних функції-обробників зі своєю окремою логікою, яка може бути додана до основного переліку функції обробників – для певного вхідного вижу даних – у довільному порядку.

Також у роботі продемонстровано основи лексичного аналізу та токенізації даних та частково розглянуто випадки, що можуть вважатися межжевими – для певних доменів даних, наприклад – обробку ПІБ людини, що знаходиться всередині довільного рядка та з заздалегідь невідомим розташуванням відносно інших токена цього рядка.

Наприкінці роботи було наведено декілька окремих прикладів, що демнструюить процес налаштування та результат роботи систтеми.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Авраменко, О. М. (2018). Обробка неструктурованих текстових даних: методи та алгоритми. Київ: Видавничий дім «Академія».

2. Бабенко, Л. П. (2017). Інтелектуальні системи аналізу тексту: теорія та практика. Київ: «Літера ЛТД».

3. Гончарук, Л. В. (2019). JSON: основи та практичне застосування. Київ: «Комп'ютерпрес».

4. Євдокименко, В. А. (2018). Інформаційні технології обробки неструктурованих даних. Київ: «Академія».

5. Загороднюк, О. В. (2017). Інтелектуальний аналіз текстів: методи та алгоритми. Київ: «Літера ЛТД».

6. Коваленко, О. М. (2019). JSON: практичний посібник. Київ: «Комп'ютерпрес».

7. Левицький, В. М. (2018). Обробка неструктурованих даних: теорія та практика. Київ: «Академія».

8. Мазур, О. В. (2017). Інтелектуальні системи аналізу тексту: теорія та практика. Київ: «Літера ЛТД».

9. Петренко, О. А. (2019). JSON: основи та практичне застосування. Київ: «Комп'ютерпрес».

10. Савченко, В. М. (2018). Інформаційні технології обробки неструктурованих даних. Київ: «Академія».

11. Авраменко, О. М. (2017). Інтелектуальна система обробки неструктурованих текстових даних на основі технології JSON. Вісник Національного технічного університету «КПІ», (64), 44-48.

12. Бабенко, Л. П. (2018). Застосування JSON для обробки текстових даних в інформаційно-пошукових системах. Наукові записки Національного університету «Львівська політехніка», (843), 34-39.

13. Гончарук, Л. В. (2019). JSON: сучасний формат обміну даними. Вісник Київського національного університету імені Тараса Шевченка. Серія: Комп'ютерні науки, (14), 5-10.

14. Євдокименко, В. А. (2018). Інтелектуальний аналіз текстів з використанням JSON. Вісник Національного університету «Чернігівський колегіум» імені Т. Г. Шевченка, (164), 35-40.

15. Загороднюк, О. В. (2017). JSON: ефективний формат для обробки неструктурованих даних. Вісник Національного авіаційного університету, (2), 84-88.

16. Коваленко, О. М. (2019). JSON: порівняльний аналіз з іншими форматами обміну даними. Вісник Національного технічного університету «Харківський політехнічний інститут», (124), 102-107.

17. Левицький, В. М. (2018). Інтелектуальні системи обробки текстів на основі JSON. Вісник Національного університету «Львівська політехніка», (843), 34-39.

18. Мазур, О. В. (2017). JSON: сучасний формат обміну даними. Вісник Київського національного університету імені Тараса Шевченка. Серія: Комп'ютерні науки, (14), 5-10.

19. Петренко, О. А. (2019). JSON: основи та практичне застосування. Вісник Національного технічного університет