May 28, 2024 Version 3

Protocol PROCI V.3

DOI
dx.doi.org/10.17504/protocols.io.261ge56qwg47/v3

Chiara Parravicini[1], Daniele Spedicati[1], Matteo Guenci[1], Nicole Liggeri[1]

[1]University of Bologna

Nicole Liggeri
University of Bologna

DOI: https://dx.doi.org/10.17504/protocols.io.261ge56qwg47/v3

Protocol Citation: Chiara Parravicini, Daniele Spedicati, Matteo Guenci, Nicole Liggeri 2024. Protocol PROCI. **protocols.io** **https://dx.doi.org/10.17504/protocols.io.261ge56qwg47/v3**Version created by **Nicole Liggeri**

Protocol status: Working
We use this protocol and it's working

Created: May 24, 2024

Last Modified: May 28, 2024

Protocol Integer ID: 100557

Keywords: analysis of peer review data, peer review data, peer reviews from crossref dataset, exchange of scholarly metadata, scholarly metadata, peer review interaction, peer review, reviewed entity, novel citation index, most peer review, citation index, cited entity, typed citation, step methodology for the systematic extraction, systematic extraction, citing entity, crossref data dump, publications in meta, publication, crossref data, extracting typed entity, crossref dataset, understanding of research impact, data analysis phase, research impact, journal, crossref, determined that the journal, entities from the same dataset, csv

Abstract

We present a step-by-step methodology for the systematic extraction, alignment, and analysis of peer review data from **Crossref** to enhance the **OpenCitations Index**.
The protocol delineates four key phases: data gathering, data processing, post-processing, data analysis and data visualization. The first two research questions were addressed during the data processing and post-processing phases, while the answers to the third, fourth, and fifth questions were found during the data analysis phase.

Purpose: The purpose of this study is to create a citation index that includes typed citations where a peer review (citing entity) reviews a publication (cited entity).
Study Design and Methodology: The study involves analyzing and manipulating the 2023 Crossref data dump. Crossref is a non-profit organization that facilitates the exchange of scholarly metadata. The outcome of the study is the development of a software capable of extracting typed entities identified as peer reviews from Crossref dataset and integrating additional information about the reviewed entities from the same dataset. This software generates CSVs and a Turtle file that conform to the OpenCitations Data Model.
Findings: Based on Crossref data, it is determined that the journal receiving the most peer reviews is PeerJ. Furthermore, it is found that approximately 5689 out of 348463 of Crossref's peer reviews are present in Meta (1.6%); 70260 are the Publications in Meta over the 77660 found in Crossref (90,5%).
Originality: This study enhances scholarly metadata and deepens the understanding of research impact by proposing a novel citation index that captures peer review interactions.

Guidelines

The required Python version for running the current software is Python 3.10.8.
Other libraries needed are:

- pytz version: 2022.7.1
- python-dateutil version: 2.8.2
- polars version: 0.20.27
- tqdm version: 4.64.1

Troubleshooting

Gathering Data

1 In order to answer to our research questions, we gathered data from **Crossref** (dump released on April 2023, zip file of 185.88GB) and **OpenCitations Meta** (dump released on April 2024, zip file of 11GB).

Dataset	
META	NAME
https://opencitations.net/download#meta	LINK

Dataset	
Crossref	NA
https://academictorrents.com/details/d9e554f4f0c3047d9f49e448a7004f7aa1701b69	L

1.1 The Crossref dump includes extensive metadata for scholarly publications. The dataset is typically organized with each record representing a single publication, containing various fields such as DOI, title, authors, publication date, journal name, and, notably, information about citations and peer reviews. Here is an example of the JSON data structure for peer reviews:

```
{
        "URL": "http://dx.doi.org/10.7554/elife.69960.sa2",
        "resource": {
            "primary": {
                "URL":
"https://elifesciences.org/articles/69960#sa2"
            }
        },
        "member": "4374",
        "score": 0.0,
        "created": {
            "date-parts": [
                [
                    2022,
                    9,
                    6
                ]
            ],
            "date-time": "2022-09-06T13:30:35Z",
            "timestamp": 1662471035000
        },
        "license": [
            {
                "start": {
                    "date-parts": [
                        [
                            2021,
                            10,
                            13
                        ]
                    ],
                    "date-time": "2021-10-13T00:00:00Z",
                    "timestamp": 1634083200000
                },
                "content-version": "unspecified",
                "delay-in-days": 0,
                "URL":
"http://creativecommons.org/licenses/by/4.0/"
            }
        ],
        "issued": {
            "date-parts": [
                [
                    2021,
```

```
                    10,
                    13
                ]
            ]
        },
        "review": {
            "type": "author-comment",
            "stage": "pre-publication"
        },
        "prefix": "10.7554",
        "reference-count": 0,
        "indexed": {
            "date-parts": [
                [
                    2022,
                    9,
                    6
                ]
            ],
            "date-time": "2022-09-06T14:14:29Z",
            "timestamp": 1662473669310
        },
        "author": [
            {
                "ORCID": "http://orcid.org/0000-0002-6672-5202",
                "authenticated-orcid": true,
                "given": "Allison",
                "family": "Schad",
                "sequence": "first",
                "affiliation": [
                    {
                        "id": [
                            {
                                "id": "https://ror.org/0130frc33",
                                "id-type": "ROR",
                                "asserted-by": "publisher"
                            }
                        ],
                        "name": "Office of Medical Education,
University of North Carolina at Chapel Hill School of Medicine",
                        "place": [
                            "Chapel Hill, United States"
                        ]
                    }
                ]
```

```
            },
            {
                "ORCID": "http://orcid.org/0000-0001-7113-1348",
                "authenticated-orcid": true,
                "given": "Rebekah L",
                "family": "Layton",
                "sequence": "first",
                "affiliation": [
                    {
                        "id": [
                            {
                                "id": "https://ror.org/0130frc33",
                                "id-type": "ROR",
                                "asserted-by": "publisher"
                            }
                        ],
                        "name": "Office of Medical Education,
University of North Carolina at Chapel Hill School of Medicine",
                        "place": [
                            "Chapel Hill, United States"
                        ]
                    }
                ]
            },
            {
                "ORCID": "http://orcid.org/0000-0001-9512-1988",
                "authenticated-orcid": true,
                "given": "Debra",
                "family": "Ragland",
                "sequence": "additional",
                "affiliation": [
                    {
                        "id": [
                            {
                                "id": "https://ror.org/0130frc33",
                                "id-type": "ROR",
                                "asserted-by": "publisher"
                            }
                        ],
                        "name": "Office of Medical Education,
University of North Carolina at Chapel Hill School of Medicine",
                        "place": [
                            "Chapel Hill, United States"
                        ]
                    }
```

```
                ]
            },
            {
                "ORCID": "http://orcid.org/0000-0003-0849-7405",
                "authenticated-orcid": true,
                "given": "Jeanette Gowen",
                "family": "Cook",
                "sequence": "additional",
                "affiliation": [
                    {
                        "id": [
                            {
                                "id": "https://ror.org/0130frc33",
                                "id-type": "ROR",
                                "asserted-by": "publisher"
                            }
                        ],
                        "name": "Office of Medical Education,
University of North Carolina at Chapel Hill School of Medicine",
                        "place": [
                            "Chapel Hill, United States"
                        ]
                    },
                    {
                        "id": [
                            {
                                "id": "https://ror.org/0130frc33",
                                "id-type": "ROR",
                                "asserted-by": "publisher"
                            }
                        ],
                        "name": "Department of Biochemistry and
Biophysics, University of North Carolina at Chapel Hill School of
Medicine",
                        "place": [
                            "Chapel Hill, United States"
                        ]
                    }
                ]
            }
        ],
        "DOI": "10.7554/elife.69960.sa2",
        "is-referenced-by-count": 0,
        "published": {
            "date-parts": [
```

```
                [
                    2021,
                    10,
                    13
                ]
            ]
        },
        "published-print": {
            "date-parts": [
                [
                    2021,
                    10,
                    13
                ]
            ]
        },
        "content-domain": {
            "domain": []
        },
        "title": [
            "Author response: Mental health in medical and
biomedical doctoral students during the 2020 COVID-19 pandemic and
racial protests"
        ],
        "source": "Crossref",
        "type": "peer-review",
        "publisher": "eLife Sciences Publications, Ltd",
        "references-count": 0,
        "deposited": {
            "date-parts": [
                [
                    2022,
                    9,
                    6
                ]
            ],
            "date-time": "2022-09-06T13:30:36Z",
            "timestamp": 1662471036000
        },
        "relation": {
            "is-review-of": [
                {
                    "id-type": "doi",
                    "id": "10.7554/eLife.69960",
                    "asserted-by": "subject"
```

```
                }
            ]
        }
    }
```

It is important to note that the capabilities of the computers at our disposal did not allow us to download the entire dump. Therefore, we leveraged more powerful servers provided by the University of Bologna to download the dataset, which subsequently has been divided into 18 chunks, each approximately 10 GB in size.

1.2 The OpenCitations Meta dump, similar to Crossref, includes detailed metadata on scholarly works. The dataset is structured to facilitate the exploration of citation networks, including peer reviews. The data typically includes fields such as the citing DOI, cited DOI, publication dates, and reviewer information. Here is an example of how data is structured in OpenCitations Meta:

	id	title	author	issue	volume	venue	page	pub_date	type	publisher	editor
	omid/0690163767doi:10.3390/inorganics	Computational Characterization Of Single-Electron Transfer Steps In Water Oxidation	De Aguirre, Adiran [omid/0624025360rcid:0000-0001-7991-6406]; Funes-Ardoiz, Ignacio [omid/0670126897171orcid:0000-0002-5843-9660]; Maseras,	3	7	Inorganics [omid/0690166425 5issn:2304-6740 openalex]	32-32	2019-03-01	journal article	Mdpi Ag [omid/0610116165 crossref:1968]	

	id	title	author	issue	volume	venue	page	pub_date	type	publisher	editor
	7030032openalex		Feliu [omid/0690494 0828]								

Data processing

2 This phase takes the Crossref data as input and selects the values from the JSON files that are necessary for alignment with the OpenCitations Index data model. The software extracts the information of peer reviews and non-peer review entities separately to generate distinct CSV files. Only the information useful according to the OC data model is extracted.

2.1 **Peer review extraction:**

During the initial setup, we use the "argparse" module to handle instructions directly from the command line. It prompts for important details such as the path of the ZIP file, where to save the output CSV files, and how many files to handle at once (given that a single chunk typically comprises 1600 files.json.gz). This configuration ensures readiness for subsequent steps. Below is a list of the required arguments:

- "zip_filename": the path to the ZIP file containing compressed JSON files.
- "output_filenames": one or more filenames for the output CSV files.
- "batch_size": number of files to process concurrently in each batch (default is 10).
- "max_files": maximum number of files to process.
- "max_workers": number of worker threads for concurrent processing (default is 2).

PeerExtractor() Class Initialization:
After the setup, the PeerExtractor class is initialized. This class takes the following parameters:

- "zip_filename": the path to the ZIP file containing compressed JSON files.

- "batch_size": the number of files to process concurrently in each batch, with a default value of 10.
- "max_workers": the maximum number of threads for concurrent processing, with a default value of 2.

"process_files" method:
With the setup complete, the script begins working through the ZIP file. This method opens the ZIP file, identifies the relevant files, and then divides them into manageable batches (the size of which is determined by the user). This method oversees the entire process, ensuring smooth operation from start to finish.
Here's a detailed list of the arguments it takes:

- "csv_writer": an instance of the CSVWriter function used for writing CSV files. This logic was implemented primarily for testing purposes because, on average, every 1600 files need CSV conversion.
- If "max_files" is specified, the function limits the number of files processed, mainly for testing purposes. After that, the function processes files in batches using the "process_batch" method.

"batch" method:
When dealing with a large number of files, often the best choice is to handle them in smaller groups. The batch method manages to divide the task into smaller parts that the script can handle more efficiently, without overloading the system, especially if one does not possess a performative machine. Here's a detailed list of the arguments it takes:

- "n": the size of each batch.

This generator method helps in splitting a large list into smaller chunks or batches, each of size n. It iterates over the input list and yields successive slices of the list, allowing for batch processing. As said, this method is useful in scenarios where processing the entire list at once would be inefficient or consume too much memory.

"process_batch" method:
Once the files are grouped together, it's time to start working on them. This method concurrently processes a given batch of files using a thread pool managed by the *ThreadPoolExecutor* module. For each file in the batch, a separate thread is created to handle the file processing, which involves reading and extracting peer review items. The results from each thread are collected and combined into a single list. The use of concurrent threads allows multiple files to be processed simultaneously, significantly speeding up the overall processing time compared to sequential processing.

"process_file" method:

Now, this method reads and unpacks each file contained in the bigger ZIP file, getting it ready for further work. This careful approach ensures each file is handled properly. Here's a detailed list of the arguments it takes:

- "zip_file": the opened ZIP file object.
- "file_to_process": the file name to process.

"process_json_data" method:
The purpose of this method can be explained as essentially decompressing and decoding the JSON data. It then attempts to parse the string as JSON, handling potential parsing errors by logging any problematic data. Once parsed, the method checks the structure of the JSON data to determine if it contains a dictionary with an "items" key or is a list directly. This logic was implemented due to the fact that the structure of the data resulted in differences between peer review information and non-peer review information. Later, the method extracts peer review items from the JSON data by filtering the relevant JSON data.

OciProcess() class initialization:
This class is responsible for preparing the necessary tools and data for converting DOIs into OCIs (Open Citation Identifiers). Here's a detailed explanation of the initialization process and the arguments it takes:

- "lookup_csv": the CSV file with character-to-code mappings (defaults to LOOKUP_CSV).
- "crossref_code": the prefix for CrossRef DOIs (defaults to CROSSREF_CODE).

During initialization, the class first reads the CSV file specified by the "lookup_csv" argument to set up a dictionary. This dictionary maps characters found in DOIs to specific codes. Next, it sets the CrossRef prefix using the "crossref_code" argument, which will be used in the conversion of DOIs to OCIs (Open Citation Identifiers). This initialization ensures that the "OciProcess" instance is equipped with the necessary data and tools to initiate the conversion process, making subsequent citation processing tasks more efficient.

"init_lookup_dic" method:
This method is responsible for setting up the lookup dictionary by reading data from a CSV file and populating it with character-to-code mappings. This is an essential part of the preparation process, ensuring the dictionary is ready for later use. Here's a detailed list of the arguments it takes:

- "lookup_csv": the path to the CSV file containing character-to-code mappings.

The method uses *csv.DictReader()* to read the CSV file, which allows for easy lookup of the mappings. It also initializes the "lookup_code" to the highest code found in the CSV. This step ensures that any new codes can be assigned sequentially without conflicts. By extracting the character-to-code mappings from the CSV file, the "init_lookup_dic" method populates the "lookup_dic" dictionary and sets the "lookup_code". This ensures the lookup dictionary is correctly populated with existing mappings and prepared to handle new characters as needed.

"calc_next_lookup_code" method:
This method basically ensures each code is unique and follows a logical order. It calculates the next available code for the lookup dictionary by first ensuring that each code is assigned sequentially and handles transitions between ranges, such as moving from 089 to 100. This systematic approach prevents duplicate codes and maintains a logical order for the codes. Here a detailed list of the arguments it takes:

"update_lookup" method:
This purpose of this method is to check if a character is already in the dictionary, and if not, it adds it with a code. By consequence its functionality is to update the lookup dictionary with a new character-to-code mapping if needed. Here a detailed list of the arguments it takes:

- "c": the character to be added to the lookup dictionary.

The purpose of this method is to check if a character, specified by "c" is already in the dictionary, and if not, it adds it with a code. The"update_lookup" method updates the lookup dictionary with a new character-to-code mapping if needed. It calculates the next available code by calling "calc_next_lookup" and appends the new mapping to the CSV file using "write_txtblock_on_csv" This ensures that the lookup dictionary remains up-to-date and consistent with the stored CSV file.

"write_txtblock_on_csv" method:
This method ensures data is stored in the right place for later use by appnding a text block to a CSV file and creating directories if necessary. Here a detailed list of the arguments it takes:

- "csv_path": the path to the CSV file.
- "block_txt": the text block to be appended.

- This method ensures data is stored in the right place for later use by appending a text block to a CSV file and creating directories if necessary. The method follows a specific flow: uses "os.makedirs" to ensure the directory exists before writing the file and handles potential errors related to directory creation, raising exceptions if issues occur. By writing a block of text to the CSV file specified by "csv_path" and ensuring

the necessary directories are created if they do not already exist, this method helps maintain persistent storage of the lookup dictionary.

"convert_doi_to_ci" method:
Now comes the main task of this section of functions: converting data. This method takes a DOI and turns it into a different format, ready for use by converting a DOI string to an OCI string using the lookup dictionary. Here a detailed list of the arguments it takes:

- "doi_str": the string of the DOI that needs to be converted

By calling the "match_str_to_lookup" converts a DOI, the "doi_str", to an OCI by translating each character in the DOI to its corresponding code using the lookup dictionary. It adds a prefix (CrossRef code) to the translated string, producing a standardized OCI. This conversion is crucial for uniquely identifying and processing DOIs in a consistent format.

"match_str_to_lookup" method:
This method checks a dictionary and assigns a code for each character, making sure everything is translated correctly. It works by converting a substring of the DOI into its corresponding code sequence. Here a detailed list of the arguments it takes:

- "str_val": a string value from which characters will be extracted and converted to corresponding codes using the lookup dictionary.

The flow of the method is the following: it translates a substring of the DOI into its corresponding code sequence using the lookup dictionary. If a character is encountered for the first time, it updates the lookup dictionary to include the new character-to-code mapping. By systematically translating each character, it generates a code sequence that uniquely represents the DOI substring. This ensures that DOIs can be consistently mapped to OCI strings for citation identification and processing purposes.

"CSVWriter" class initialization
Before writing data into CSV files, there's a setup phase. This part of the code prepares the CSVWriter class. It's like getting the right tools ready before starting work. The class is designed to handle the output filenames. If there's only one filename provided, it stores it as a list. Otherwise, it keeps them as they are. This ensures that the class can handle both single and multiple output files efficiently. Here a detailed list of the arguments it takes:

- "output_filenames": The filename(s) for the output CSV file(s).

"write_to_csv" method:

Once everything is set up, it's time to write data into the CSV files. This method opens each output file and starts putting data into them. For each piece of data, it creates a row in the CSV file. Here a detailed list of the arguments it takes:

- "peer_review_items": List of peer review items to be written to CSV.

This process repeats until all data is written. Additionally, this method is responsible for creating a unique identifier for each peer review relationship using the Open Citation Identifier (OCI) format. It ensures that each relationship between citing and cited entities has a distinct identifier. This helps in organizing and referencing the peer review data accurately.

"remove_duplicates" method:

After writing data, there's the cleanup step. This method ensures that there are no duplicate entries in the CSV files. It reads the CSV file, removes any duplicate rows based on the OCI (Open Citation Identifier), and saves the cleaned-up data into a new file. Here a detailed list of the arguments it takes:

- "input_filename": the name of the input file
- "output_filename": the name of the output file

This method basically ensures that the final CSV files contain unique peer review items, avoiding any redundancy or repetition.

Main function argument, parsing, and execution flow:
The main function is responsible for parsing command-line arguments, initializing necessary components, and coordinating the entire processing workflow. Here's a breakdown of its flow:

- Initializes CSVWriter and NonPeerExtractor instances.
- Processes JSON files and writes peer review items to CSV.

In essence, the main function acts as the captain steering the ship. It listens for commands, prepares the necessary tools (CSVWriter and PeerExtractor), and initiates the processing workflow. This involves processing JSON files, writing peer review data to CSV files, and removing duplicates from the final output. By orchestrating these tasks, the main function ensures a smooth and organized execution of the entire process. It was chosen during development to save two CSV files in the peer review items extraction phase, one which is not altered and one which is filtered from duplicates as described above. Ths choice was made because in this way the final user can safely decide which file to use for his/her purposes.

The extractions of items related non peer review entries follow the exact same logic, of course, with some modification, mainly in the fact that the "type" key in the JSON files in which the interest falls on in this phase is not anymore "peer-review" but everything else. The second and last modification to the logic is that everything regarding the OCI creation and its related implementations is not in the "NonPeerExtractor.py" software because there was non need to calculate the OCI.

It is necessary to mention that the class "OciProcess" and all its relative methods were taken from the COCI workflow contained in **this Github** page.

3 Filtering, joining and calculation of delta for the CSVs

3.1 Initial setup and argument parsing:

The script starts by parsing the command-line arguments using the argparse module. It expects three required arguments. This setup phase ensures that the script receives the necessary input to perform its tasks. Here a detailed list of the arguments it takes:

- "peer_review_dir": Directory containing peer-reviewed CSV files.
- "non_peer_review_dir": Directory containing non-peer-reviewed CSV files.
- "output_path": Path for the output CSV file.

Class initialization:

The "Filter" class is then initialized with the provided directories for peer-reviewed and non-peer-reviewed data, the output path, and the column to join on ("cited_doi").

Reading and Concatenating DataFrames:

The "read_and_concatenate_dataframes" method reads CSV files from the specified directory and then for each file, it uses "polars.scan_csv" to read the file and normalizes the "cited_doi" column by stripping newline and period characters and converting to lowercase. DataFrames are concatenated using "polars.concat". Here a detailed list of the arguments it takes:

- "directory": the directory from which it needs to read the files.

During the developmente of this method the "polars.scan_csv" function was used, mostly due to the fact that it lazily read CSV files, meaning that It does not load the entire file into memory, which is efficient for large datasets. Here's a deeper explanation:

- It scans the file and allows for subsequent operations to be performed as a pipeline creating a LazyFrame Object that represents a deferred execution plan rather than an actual dataset in memory. It's designed to optimize memory usage by delaying computation until absolutely required..

After that the normalization process ensures that DOI values are consistent by removing unwanted characters (from the start and the end of the string) and converting them to lowercase, which is important for accurate joins.

Validating DataFrames:
Before proceeding further, the script validates the DataFrames to ensure that the column to join on ("cited_doi") exists in both DataFrames. This validation step prevents runtime errors during the join operation. Here a detailed list of the arguments it takes:

- "df1": the first of the two DataFrames that the method is comparing
- "df2": the second of the two DataFrames that the method is comparing

Joining DataFrame:
Then, the join_dataframes method of the Filter class performs an inner join on the two DataFrames using the "cited_doi" column. This join operation combines the peer-reviewed and non-peer-reviewed data based on matching DOI values, creating a unified dataset that will be later used for further analysis. Here a detailed list of the arguments it takes:

- "df1": the first of the two DataFrames that the method is comparing
- "df2": the second of the two DataFrames that the method is comparing

Adding provenance information:
The "add_provenance method" add to the joined DataFrame with provenance information. It adds three new columns: "prov_agent" (URL indicating the provenance agent), "source" (URL of the data source), and "prov_date" (current timestamp in UTC). Here a detailed list of the arguments it takes:

- "df" the Dataframe on which to add the specified columns

These additional information are useful to track the origin and processing history of the data, ensuring transparency and reproducibility.

Calculating time delta:
The "Delta" class is then initialized with its provided DataFrame. It comprehends different static methods which don't depend on specific object instances. They do not access or modify instance-specific attributes, but rather operate solely on the parameters passed to them. Here a detailed list of the arguments it takes:

- "df": the DataFrame on which the class will perform its calculations (the joined DataFrame)

Here, instead a list of the static methods:

- "contain_years": This method checks if the length of the date string ("date_str") is at least 4 characters long. In most date formats, the year is represented by 4 digits, so if the length of the string is 4 or greater, it likely contains information about the year.
- "contain_months": this method checks if the length of the date string is at least 7 characters long. In typical date formats, if the length is 7 or greater, it indicates that the string likely includes information about both the year and the month.
- "contain_days": This method checks if the length of the date string is at least 10 characters long. In common date representations, such as "YYYY-MM-DD", the length of the string is 10 characters if it includes information about the year, month, and day.

These methods are responsible for the validation of date strings by determining the level of detail they contain.

The "calculate_date_difference" method:
Checks if dates are valid, basically if they have years, months and days specified. Then parses the dates and calculates the difference using "dateutil.relativedelta". Then finally it formats the difference as an ISO 8601 duration string. Here a detailed list of the arguments it takes:

- "citing_date": the date of the citing entity (the peer review)
- "cited_date": the date of the cited entity (the article reviewed)

More in details: "dateutil.relativedelta" is responsible for providing a way to compute the difference between two dates considering years, months, and days, which is more precise than simple subtraction. ISO 8601 duration is instead responsible for managing the standard format for date and time durations, it basically ensures that the output is universally understandable and machine-readable.

The "add_delta_column" method:
This method contained in the "Delta" class adds a new column to the DataFrame that represents the time difference between "citing_date" and "cited_date" in each row. Thismethod takes no argument because it is called directly on the joined DataFrame. The process involves the use of "pl.struct" and pl.map_elements" functions provided by the Polars library. Here's a deeper explanation of these functions:

- "pl.struct": This function is used to combine multiple columns into a single struct column. In this case, it takes the "citing_date" and "cited_date" columns and creates a struct that groups these two columns together for each row. By combining these two date columns into a struct, it allows subsequent operations to treat them as a single

entity. This is particularly useful for applying functions that require multiple inputs, such as calculating the time difference between two dates.

- "pl.map_elements": This function is applied to the struct column created in the previous step. It maps a given function to each element (each struct) of the column. In this context, it applies the "calculate_date_difference" function to each struct containing "citing_date" and "cited_date". The "map_elements" function allows for element-wise operations on struct columns. It takes each struct (which contains a pair of dates) and processes it with the provided function to compute the desired output.
- Lambda function: The lambda function contained in this method, "lambda x:self.calculate_date_difference(x["citing_date"], x["cited_date"])" is used to pass the dates from each struct to the "calculate_date_difference" method.
- The "return_dtype=pl.Utf8" specifies that the result of the function (the time span) will be of string type.
- Creating the "time_span" Column: The .alias("time_span") part renames the resulting column to time_span, which holds the computed time differences.

The "save_csv" method:
This method is responsible for saving the final DataFrame (with provenance and delta columns) to the specified output path. Here a detailed list of the arguments it takes:

- "output_csv": the desired name for the output CSV

This method uses "polars.sink_csv" function which is determined by the choice of using "plars.scan_csv" function before. Basically using the scan function a LazyFrame Object is created which is more efficient in terms of memory than a DataFrame object, particularly for large datasets, as it postpones the execution of operations until necessary. However, due to its lazy evaluation nature, direct usage of "write_csv" function becomes inaccessible.

- "sink_csv": This method is used to write the DataFrame to a CSV file efficiently. it efficiently serializes and writes tabular data from a DataFrame or LazyFrame Object to a CSV file on disk. It is part of the Polars library, which is optimized for performance and can handle large datasets effectively.

Main execution flow:
Finally, the main function orchestrates the entire process, executing each step in sequence. It reads, concatenates, validates, and joins the DataFrames, adds provenance information, calculates the time delta, and saves the final CSV file with the delta column. This structured execution flow ensures that each task is performed accurately and

efficiently, resulting in a comprehensive analysis of the input data. Here the flow of the calling done by the "main" function:

- Parses arguments and initializes the Filter class.
- Reads, concatenates, and validates DataFrames.
- Joins DataFrames and adds provenance information.
- Initializes Delta class and calculates the time_span column.
- Saves the resulting DataFrame to the output path.

Post-processing

4 The main purpose of this phase of the worklflow is using the processed data about peer reviews and articles reviewed we obtained as output of the previous steps to create instances of the class PeerReview and to generate a RDF graph. These operations satisfy the requests contained in the first two research questions (**RQ1, RQ2**), which focus on the creation of typed citations and the creation of an index compliant to the OpenCitations Data Model.
In this phase we also save information retrieved in the previous steps in different CSV files for convenience purposes, as explained in the following section.

4.1 **Compartimentizing data into separate CSV files**
The Compartimentizer class has been designed to process the CSV file obtained as output of the classes Filter and Delta; the Compartimentizer splits the CSV's columns into three separate CSV files: one containing information about citations (storing oci, citing doi and url, cited doi and url, citing date, cited date and timespan), one about provenance (storing oci, agent, source, creation date), one about venues (storing venues and ISSN codes). This operation can be useful for organizing and distinguishing different parts of the data, especially if it is to be included in a larger dataset.

1. **Initialization (__init__ method):**
   - The Compartimentizer class is initialized with parameters specifying which columns to drop and where to save the output CSV files.
   - The parameters are stored as attributes of the instance.
2. **Compartmentalizing Data (compartimentizer method):**
   - **Reading Input File:** the input CSV file is read into a Polars DataFrame (df) using pl.read_csv(path).
   - **Cloning DataFrames:** two additional DataFrames (df1 and df2) are created by cloning the original DataFrame to retain the complete data structure for separate processing.
   - **Dropping Columns:** columns specified in columns_to_drop are dropped from df to create the first output DataFrame; columns specified in columns_to_drop1 are dropped from df1 to create the second output DataFrame; columns specified in columns_to_drop2 are dropped from df2 to create the third output DataFrame.

- **Writing Output Files:** the resulting DataFrames are written to their respective output paths using write_csv.

4.2 **Creating PeerReview entities and updating the RDF**

The PeerReview class and its supporting functions provide a framework for transforming CSV-based citation data into RDF format compliant with the OpenCitations Data Model. This process facilitates the creation of a new index of citations, enabling analysis of peer reviews and their relationships with scholarly publications.
Similarly to what has been specified for the OciProcessor class, also for shaping the PeerReview class and its supporting functions the code was readapted from the COCI workflow contained in **this Github** page.

1. **Initialization of PeerReview Object:**
   - An instance of the PeerReview class is created for each row in the CSV file. The __init__ method extracts the relevant data from the row and initializes the object.
2. **RDF Graph Generation (get_peer_review_rdf):**
   - The get_peer_review_rdf method creates an RDF graph for each PeerReview object.
3. **Data Inclusion:**
   - Adds triples for the citing and cited entities using the citing_url and cited_url.
   - Adds triples for citation creation date and time span using the citing_date and time_span.
4. **Provenance Inclusion:**
   - Adds triples for provenance information using prov_agent_url, source, and prov_date.
5. **Data Population (populate_data):**
   - The populate_data function reads the input CSV and creates PeerReview objects for each row.
   - It generates RDF graphs for each object and writes the serialized graphs to the output file.
6. **Provenance Population (populate_prov):**
   - The populate_prov function is similar to populate_data but focuses on provenance data.
   - It reads the input CSV, creates PeerReview objects with provenance information, generates RDF graphs, and writes the serialized graphs to the output file.
7. **Output:**
   - The output files are in N-Triples format, containing RDF triples that represent peer review data and provenance information. These files can be used for further analysis or integration into other RDF-based systems.

Data analysis

5 The data analysis phase of our project focuses on extracting relevant insights from the processed data obtained from Crossref; furthermore, a comparison with the database of

OpenCitations Meta is performed, cross-referencing DOIs (Digital Object Identifiers) from our combined dataset with the OpenCitations Meta dataset to quantify the overlap between peer reviews and articles.

5.1 **Extracting the top publication venues**

The VenueCounter class is designed to process the CSV file obtained as output of the Filter and Delta classes in order to retrieve information about citations and their associated venues, handling various cases of ISSN and venue information availability. The goal is to count and identify the top publication venues in terms of the number of peer reviews received, as asked in our third research question (**RQ3**).

1. **Initialization (__init__ method):**
   - The VenueCounter class is initialized with the path to the input CSV file. This path is stored in the csv_file_path attribute.
2. **Counting Venues (count_venues method):**
   - **Loading CSV File: t**he CSV file is read into a pandas DataFrame using specified data types for the cited_issn and cited_venue columns; missing values in these columns are replaced with empty strings.
   - **Processing Multiple ISSNs:** rows containing multiple ISSN values (comma-separated) are identified and split into separate columns (issn1 and issn2); these rows are grouped by issn1, issn2, and cited_venue, and their occurrences are counted.
   - **Processing Single ISSN:** rows without commas in cited_issn are grouped by cited_issn and cited_venue; the cited_issn column is renamed to issn1, and an issn2 column with None values is added; these grouped rows are counted.
   - **Processing Empty ISSN:** rows with empty cited_issn are grouped by cited_venue; columns issn1 and issn2 are added with appropriate values (issn1 as empty string and issn2 as None); these grouped rows are counted.
   - **Combining Results:** the grouped DataFrames from the three cases are concatenated into a single DataFrame; duplicate rows are dropped, ensuring unique combinations of ISSNs and venues.
   - **Returning Final DataFrame:** the final grouped DataFrame is returned.
3. **Getting Top Venues (get_top_venues method):**
   - The count_venues method is called to get the grouped DataFrame.
   - The DataFrame is sorted by the count of occurrences in descending order.
   - The top n venues (default is 10) are returned.
4. **Saving Results to CSV (save_to_csv method):**
   - The count_venues method is called to get the grouped DataFrame.
   - The DataFrame is saved to the specified output CSV file.
   - A confirmation message is printed indicating the location of the saved file.

5.2 **Comparing Crossref data with OpenCitations Meta**

The MetaAnalysis class is designed to process an OpenCitations Meta dump (contained in a zip file) to extract DOIs and compare them against lists of peer reviews and articles reviewed we retrieved from the Crossref dump; the results are saved to a CSV file. The

goal is to determine the number of peer reviews and articles in Crossref that are also included in OpenCitations Meta, as asked in our fourth and fifth research question (**RQ4, RQ5**).

1. **Initialization (__init__ method):**
   - The MetaAnalysis class is initialized with the path to the combined CSV file, which is stored in the combined_csv_path attribute.
2. **Extracting DOIs from Meta File (extract_doi_from_meta method):**
   - **Opening Output File:** the output file is opened for writing, and the header 'DOI' is written.
   - **Processing Zip File:** the zip file is opened, and CSV files within it are identified; for each CSV file, the content is read in chunks, and DOIs are extracted using the extract_doi_from_text method; the extracted DOIs are written to the output file, excluding empty values.
3. **Extracting DOI from Text (extract_doi_from_text method):**
   - **Finding DOI:** the method checks if the text contains 'doi:'; if found, it extracts the substring starting after 'doi:' and ending at the next space or end of text; if the extracted DOI contains a comma, it is enclosed in quotes; the extracted DOI is returned, or an empty string if no DOI is found.
4. **Counting Rows (count_rows method):**
   - The method opens the specified file and counts the number of lines, subtracting one for the header to get the total row count.
5. **Creating Peer Review and Article Lists (create_peer_review_and_article_lists method):**
   - **Reading Combined CSV:** the combined CSV file is read into a DataFrame.
   - **Creating DOI Sets:** sets of DOIs for peer reviews (citing_doi) and articles (cited_doi) are created.
   - **Processing Meta File:** the meta file is read in chunks, and DOIs matching the peer review and article sets are identified; the matching DOIs are written to separate output files (meta_peer.csv and meta_article.csv).
6. **Dropping Duplicates and Saving (drop_duplicates_and_save method):**
   - The input file is read into a DataFrame, duplicate rows are dropped, and the cleaned data is saved to the output file.
7. **Getting Peer Review Count (get_peer_review_count method):**
   - The method calls extract_doi_from_meta to extract DOIs from the meta file.
   - Calls create_peer_review_and_article_lists to create peer review lists.
   - Calls drop_duplicates_and_save to clean the peer review list.
   - Calls count_rows to count unique peer review DOIs and returns the count.
8. **Getting Article Count (get_article_count method):**
   - The method calls extract_doi_from_meta to extract DOIs from the meta file.
   - Calls create_peer_review_and_article_lists to create article lists.
   - Calls drop_duplicates_and_save to clean the article list.
   - Calls count_rows to count unique article DOIs and returns the count.

9. **Saving Counts to CSV (save_counts_to_csv method):**
   - The method creates a DataFrame with the counts of peer reviews and articles.
   - Writes the DataFrame to the specified output file.
   - Prints a confirmation message indicating the path to the saved file.

Visualizing results

6 Data visualization has been realized using the Python library Matplotlib (**https://matplotlib.org/**); the following graphs have been created as a result of the analysis performed:

- a vertical bar chart showing the number of typed citations (i.e. a peer review reviews a publication) created, the number of peer reviews and the number of reviewed bibliographic resources retrieved from the Crossref dump;
- a horizontal bar chart showing the top ten venues ordered by the number of reviews received by the resources the venues contain;
- two donut charts displaying how many citing and cited DOIs retrieved from Crossref are also contained in the OpenCitations Meta dataset;
- a line chart showing the number of bibliographic resources that received a certain number of peer-reviews in Crossref;
- a horizontal bar chart reporting the top ten cited DOIs in Crossref;
- a donut chart showing the percentages of positive and negative timespan between the publication of a bibliographic resource and the related peer review;
- two horizontal bar charts showing the top ten venues by number of citations with a negative timespan and the number of citations received by venues without a reported ISSN;
- a vertical bar chart displaying the number of cited resources in Crossref, the number of cited resources that are also contained in Meta and the number of cited resources, present both in Crossref and Meta, whose citing resource is also present in Meta.

These graphs can be found, with their related descriptions and data anlaysis report, in the article reporting the results of this research (available in the Materials section of this protocol).

Protocol references

Heibi, I., Peroni, S. & Shotton, D. Software review: COCI, the OpenCitations Index of Crossref open DOI-to-DOI citations.Scientometrics 121, 1213–1228 (2019). https://doi.org/10.1007/s11192-019-03217-6.