FDup framework: A General-purpose solution for Efficient Entity Deduplication of Record Collections*

Michele De Bonis[1,*,†], Claudio Atzori[1], Sandro La Bruzzo[1] and Paolo Manghi[1,2]

[1]*Consiglio Nazionale delle Ricerche - Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo" (ISTI-CNR), Pisa, Italy*
[2]*OpenAIRE AMKE, Marousi (Athens), Greece*

Abstract

Deduplication is a technique aimed at identifying and resolving duplicate metadata records in a collection with a special focus on the performances of the approach. This paper describes FDup(Flat Collections Deduper), a general-purpose software framework supporting a complete deduplication workflow to manage big data record collections: metadata record data model definition, identification of candidate duplicates, identification of duplicates. FDup brings two main innovations: first, it delivers a full deduplication framework in a single easy-to-use software package based on Apache Spark Hadoop framework, where developers can customize the optimal and parallel workflow steps of blocking, sliding windows, and similarity matching function via an intuitive configuration file; second, it introduces a novel approach to improve performance, beyond the known techniques of "blocking" and "sliding window", by introducing a smart similarity-matching function T-match. T-match is engineered as a decision tree that drives the comparisons of the fields of two records as branches of predicates and allows for successful or unsuccessful early exit strategies. The efficacy of the approach is proved by experiments performed over big data collections of metadata records in the OpenAIRE Graph, a known open-access knowledge base in Scholarly communication.

Keywords

Data Disambiguation, Scholarly Communication, Deduplication

1. Background

Deduplication is a technique for the identification and purging of duplicate metadata records in large datasets, addressing challenges like efficiency and flexibility in the wider context of entity resolution[1, 2, 3, 4, 5]. As emerged from the surveys on this topic[6, 7], traditional methods use: (i) a preliminary blocking phase to group potentially equivalent records, (ii) sliding window techniques to manage the workload and perform pair-wise similarity matches, and (iii) a final transitive closure to create groups of duplicates. The paper briefly introduces FDup (Flat Collections Deduper), originally presented in [8] and built on Apache Spark, enhancing deduplication by offering customizable configurations and efficient similarity-matching through a decision tree-driven approach. FDup has been conceived as an evolution of the GDup framework[9]

SEBD 2024: 32nd Symposium on Advanced Database Systems, June 23-26, 2024, Villasimius, Sardinia, Italy

[*] Corresponding author.

[†] These authors contributed equally.

michele.debonis@isti.cnr.it (M. De Bonis); claudio.atzori@isti.cnr.it (C. Atzori); sandro.labruzzo@isti.cnr.it (S. La Bruzzo); paolo.manghi@openaire.eu (P. Manghi)

0000-0003-2347-6012 (M. De Bonis); 0000-0001-9613-6639 (C. Atzori); 0000-0003-2855-1245 (S. La Bruzzo); 0000-0001-7291-3210 (P. Manghi)

and is proven to improve its performance in handling flat record collections. The framework is successfully deployed in the OpenAIRE Graph to manage over 300M records, demonstrating significant usability and performance benefits in real-world big data scenarios.

Outline: Section 2 formally presents the functional architecture of FDup, focusing on the requirements, the model adopted, and the technical implementation of the framework, providing an example of its usage in the OpenAIRE infrastructure. Section 3 describes methods and techniques used to test both the framework efficiency and usability defining an innovative custom configuration for the deduplication. Section 4 provides experimental results and highlights how FDup overcomes traditional approaches for time consumption. Section 5 conclude the paper and delve into possible future works and developments of the framework.

2. Software description

2.1. Architecture

FDup realizes the deduplication workflow shown in Figure 1. The workflow is intended to deduplicate large collections of records, processing them in four sequential phases:

- *Collection import*, to set the record collection ready to process by mapping into a “flat” record with attributes (the use case of this paper is the publication deduplication, using PIDs, title, and authors list);
- *Candidate identification*, to block the records to be matched by sorting them following an $orderField$ and scanning pairs using a sliding window mechanism;
- *Duplicates identification*, to efficiently identify pairs of equivalent records via the T-match function;
- *Duplicates grouping*, to identify groups of equivalent records via transitive closure.

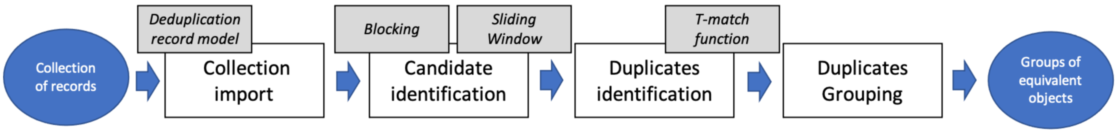

Figure 1: FDup deduplication workflow

The majority of deduplication frameworks in the literature encode record similarity-matching conditions via a similarity function of the form:

$$f([v_1, \dots, v_k], [v'_1, \dots, v'_k]) = \sum_{i:0\dots k} f_i(v_i, v'_i) \times w_i$$

where the v_i’s are the values of field l_i, $f_i(v_i, v'_i)$ are *comparators*, functions measuring the “distance” of v_i and v'_i for the field l_i, and w_i’s are the weights assigned to the comparators f_i’s, such that $\sum_{i:0\dots k} w_i = 1$. As a result, f returns a value in a given range, e.g. $[0 \dots 1]$, scoring the “distance” between two records. The records are considered equivalent if the distance measure exceeds a given threshold.

For the example above, the similarity function $PublicationWeightedMatch$, created using the GDup framework in OpenAIRE, encodes both equivalence by identity and by value as follows:

$$\begin{aligned} PublicationWeightedMatch(r, r') = \; & jsonListMatch(r.PIDs, r'.PIDs) \times 0.5+ \\ & TitleVersionMatch(r.title, r'.title) \times 0.1+ \\ & AuthorsMatch(r.authors, r'.authors) \times 0.2+ \\ & LevenshteinTitle(r.title, r'.title)) \times 0.2 \end{aligned}$$

where $jsonListMatch$, applied to the field PID, returns 1 if there is at least one PID in common in the two records; $TitleVersionMatch$, applied to the titles, returns 1 if the two titles contain identical numbers or Roman numbers; $LevenshteinTitle$ returns 1 if the two (normalized) titles have a Levenshtein distance greater than 90%, and $AuthorsMatch$ performs a “smart” matching of two lists of author name strings and returns 1 if they are 90% similar (the minimal equivalence threshold is computed over a manually validated ground truth of equivalent records). All comparators return 0 if their condition is not met. The minimal threshold for two records to be equivalent is 0.5, the threshold that can be reached by $jsonListMatch$ alone or by combining the positive results of the three functions $TitleVersionMatch$, $AuthorsMatch$, and $LevenshteinTitle$. All f_i’s in $PublicationWeightedMatch$ are computed and averagely require a constant execution time, despite the successful or unsuccessful match that those may feature.

Motivated by such observation, FDup introduces a similarity match function T-match that returns an equivalence match exploiting a decision tree, nesting the comparator functions. Each tree node verifies a condition, which can be the result of combining one or more comparators, and introduces a positive (*MATCH*) or negative (*NO_MATCH*) *exit strategy*. If the exit strategy is not fired, T-match heads to the next node. An early exit skips the full traversal of the tree and can turn the result into a *MATCH*, i.e., a *simRel* relationship between the two records is drawn, or into a *NO_MATCH*, i.e., no relationship is drawn. By doing this, T-match allows to save time by avoiding unnecessary computations.

A T-match decision is formed by a tree of *named nodes* with outgoing *edges*. The core elements of a T-match node are the *aggregation* function, the list of *comparators*, and a *threshold* value. The aggregation function collects the output of the comparators and delivers an “aggregated” result based on one of the following functions: maximum, minimum, average, and weighted mean. The execution of a T-match node must end with a decision, which may be:

- *positive*, i.e. the result of the aggregation function is greater or equal to the threshold value;
- *negative*, i.e. the result of the aggregation function is lower than the threshold;
- *undefined*, i.e. one of the comparators cannot be computed (e.g. absence of values); a node also bears a flag *ignoreUndefined* that ignores the *undefined* edge even if one of the values is absent.

For each decision, the node provides the *name* of the next node to be executed. By default, T-match provides two nodes *MATCH* and *NO_MATCH* to be used to force a successful or unsuccessful early exit from the tree.

The example in Figure 2 shows the function $PublicationTreeMatch$, which uses the same comparators but exploits a T-match decision tree. The individual matches are lined up by introducing *MATCH* conditions early in the process, i.e. equivalence by identity via $PIDMatch$, and then ordering *NO_MATCH* conditions by ascendant execution time, i.e. equivalence by value via $versionMatch$, $titleMatch$, and $authorsMatch$.

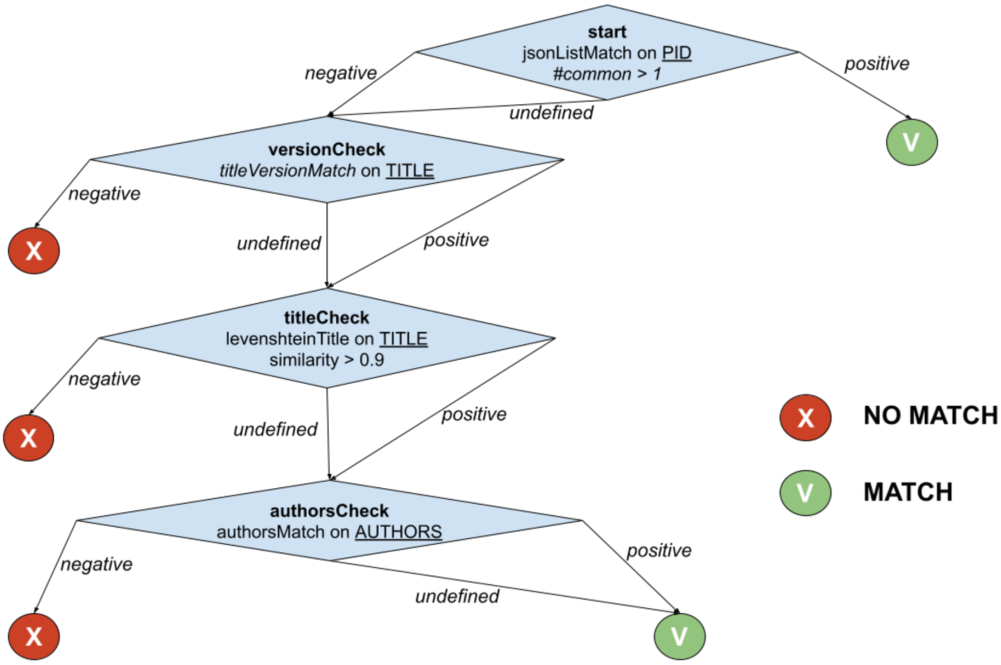

Figure 2: T-match's decision tree for $PublicationTreeMatch$: (i) compute the $jsonListMatch$. A *simRel* relationship is drawn if there is at least 1 PID in common, otherwise it proceeds to the next node; (ii) compute the $TitleVersionMatch$. The computation is interrupted if the titles do not contain identical numbers, otherwise it proceeds to the next step; (iii) compute the $LevenshteinTitle$. The computation is interrupted if the Levenshtein distance is lower than 90%, otherwise it proceeds to the next step; (iv) compute the $AuthorsMatch$. A *simRel* relationship is drawn if the authors' lists are at least 90% similar.

T-match allows for the definition of multiple paths, hence the simultaneous application of alternative similarity match strategies in one single function. The experiments described in later sections will show that when the number of records is large, T-match significantly improves the overall performance of the deduplication process.

2.2. Implementation

FDup's software is structured in three modules, `Pace_Core`, `Dedup_Workflow`, and

`Configuration file` depicted in Figure 3. The framework is implemented in Java and Scala, and grounds on the Apache Spark Framework, an open-source distributed general-purpose cluster-computing framework. FDup exploits Apache Spark to define record collection parallel processing strategies that distribute the computation workload and reduce the execution time of the entire workflow. Scala is instead required to exploit the out-of-the-box library for the calculation of a "closed mesh" in GraphX[1]. The software is published in Zenodo.org by [10].

The three modules implement the following aspects of FDup's architecture:

- `Pace_Core` includes the functions implementing the candidate identification phase (blocking and sliding window) and the T-match function, as well as the (extensible) libraries of comparators and clustering functions.
- `Dedup_Workflow` is the code required to build a deduplication workflow in the Apache Spark Framework by assembling the functions in `Pace_core` according to the comparators, clustering functions, and parameters specified in the `Configuration file`.
- `Configuration file` sets the parameters to configure the deduplication workflow steps, including record data model, blocking and clustering conditions, and T-match function strategy.

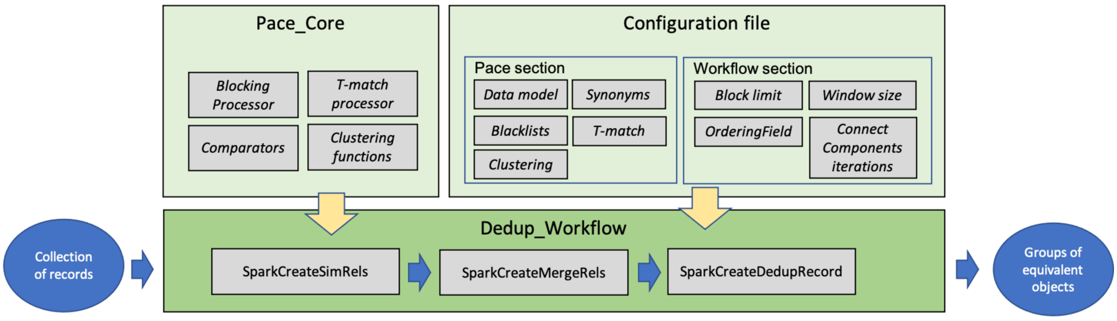

Figure 3: FDup software modules

The deduplication workflow is implemented as an Oozie workflow that incapsulates jobs executing the three steps depicted in Figure 3, to compute: (*i*) the similarity relations ($SparkCreateSimRels$), (*ii*) the merge relations ($SparkCreateMergeRels$), and (*iii*) the groups of duplicates and the related representative objects ($SparkCreateDedupEntity$). More specifically:

- $SparkCreateSimRels$: uses classes in the `Pace_Core` module to divide entities into blocks (clusters) and subsequently computes $simRels$ according to the `Configuration file` settings for T-match;
- $SparkCreateMergeRels$: uses GraphX library to process the $simRels$ and close meshes they form; for each connected component, a master record ID is chosen and $mergeRels$ relationships are drawn between the master record and the connected records;
- $SparkCreateDedupEntity$: uses $mergeRels$ to group connected records and create the representative objects.

[1] Apache Spark GraphX, https://spark.apache.org/graphx/

3. Experiments

The experiment aims to show the performance gain yielded by the proper configuration of T-match in a deduplication workflow for the publication similarity match example presented in Figure 2. To this aim, the experiment sets two deduplication workflows with identical blocking and sliding window settings but distinct similarity-matching configurations. Both configurations address the similarity criteria but in opposite ways:

- $PublicationTreeMatch$ **configuration**: a configuration that implements the $PublicationTreeMatch$ decision tree illustrated in Figure 2, taking advantage of early exits and the T-match ;
- $PublicationWeightedMatch$ **configuration**: a configuration that implements the similarity match as the GDup (average mean) function $PublicationWeightedMatch$ described in Section 2.1 by combining all comparators in one node, whose final result is a *MATCH* or *NO_MATCH* decision.

Both configurations are based on the same settings for candidate identification and duplicate identification. In particular:

- The clustering functions used to extract keys from publication records are the $LowercaseClustering$ on the DOI (e.g. a record produces a key equal to the lowercase DOI, the result is a set of clusters composed by publications with the same DOI) and the $SuffixPrefix$ on the publication title (e.g. a record entitled *"Framework for general-purpose deduplication"* produces the key *"orkgen"*, the result is a set of clusters composed by publications with potentially equivalent titles); both functions are described in **??**
- the $groupMaxSize$ is set to 200 (empirically) to avoid the creation of big clusters requiring long execution time;
- the $slidingWindowSize$ to limit the number of comparisons inside a block is set to 100 (empirically).

Both $PublicationTreeMatch$ and the $PublicationWeightedMatch$ configurations were performed over the publication record collection published in [11]. The collection contains a set of 10M publications represented in JSON records extracted from the OpenAIRE Graph Dump[12]. In particular, publications have been selected from the Dump to form a dataset with a real-case duplication ratio of around 30% and an appropriate size to prove the substantial performance improvement yielded by the early exit approach.

Two tests were performed, comparing the performance of the configurations $PublicationTreeMatch$ and $PublicationWeightedMatch$ over the 10M and the 230M collection respectively. The tests are intended to measure the added value of T-match in terms of performance gain, i.e. $PublicationTreeMatch$ vs $PublicationWeightedMatch$ execution times.

The tests were performed with a driver memory set to 4 Gb, the number of executors to 32, the executor cores to 4, and the executor memory to 12 Gb. The Spark dynamic allocation has been disabled to ensure a fixed amount of executors in the Spark environment, so as to avoid aleatory behavior. Moreover, since Spark's parallelization shows different execution times, depending on both the distribution of the records in the executors and the cutting operations on the blocking phase, each test has been executed 10 times and the average time has been calculated.

The execution time was measured in terms of processing time required by $SparkCreateSimRels$, where the pair-wise comparisons are performed, and by $SparkCreateMergeRels$, where groups of duplicates are generated. It was observed that $SparkCreateSimRels$ is dominant taking 70% of the overall processing time. As a consequence, for the sake of experiment evaluation, we: (i) reported and confronted the time consumed by $SparkCreateSimRels$ under different tests to showcase the performance gain of T-match, and (ii) reported the results of the $SparkCreateMergeRels$ to ensure that the tests are sound, i.e. yield the same number of groups, or with a small relative change percentage due to the aleatory behavior described above.

4. Results

The results of the tests on the 10M publication records dataset and the 230M full publication datasets are depicted in Figure 4 and Figure 5, respectively. The graphs show the average time consumption of the $SparkCreateSimRels$ phase for each execution of the test.

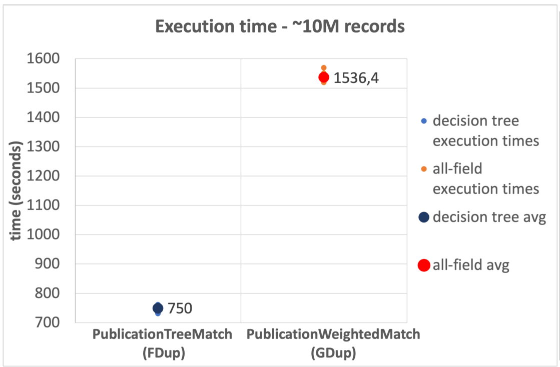

Figure 4: 10M records test

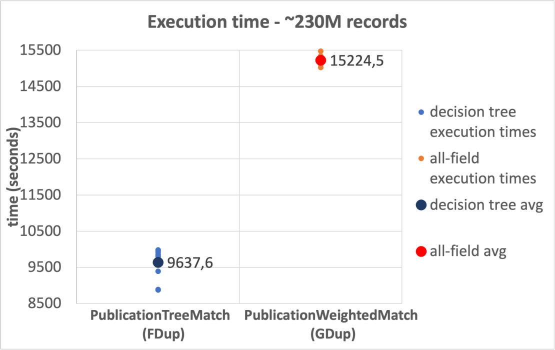

Figure 5: 230M records test

The average time of the $SparkCreateSimRels$ stage in the test performed over 10M records dataset with the $PublicationTreeMatch$ configuration is 750 seconds, while the $PublicationWeightedMatch$ configuration consumes $1,536.4$ seconds. The $SparkCreateSimRels$ test on the 230M records dataset features an average time of $9,637.6$ seconds for $PublicationTreeMatch$ and of $15,224.5$ seconds for $PublicationWeightedMatch$.
The results reported in Table 1 show that the two scenarios produced a comparable but not identical amount of $simRels$, $mergeRels$, and $connectedComponent$ due to the aleatory behavior Spark.

Based on such results, it can be stated that the $PublicationTreeMatch$ configuration overtakes the $PublicationWeightedMatch$ configuration in terms of time consumption, by improving performance up to 50% in the first test and up to 37% in the second test.

Table 1
Average number of relations drawn by the deduplication workflow on 10M and 230M publication records

size	relation type	TreeMatch	WeightedMatch	relative change (%)
10M	$simRels$	13,865,552	13,866,320	0.000055
	$mergeRels$	5,247,252	5,247,585	0.000063
	$connectedComponents$	1,890,012	1,890,148	0.000071
	$pairwiseComparisons$	255,772,628	255,772,628	0.0
230M	$simRels$	172,510,072	172,511,772	0.0000098
	$mergeRels$	69,974,139	69,974,155	0.00000022
	$connectedComponents$	25,250,036	25,250,143	0.0000042
	$pairwiseComparisons$	3,650,733,202	3,650,733,202	0.0

5. Conclusions

This work presented FDup, a framework for the deduplication of record collections that allows: (i) to easily and flexibly configure the deduplication workflow depicted in Figure 1 and (ii) to add to the known execution time optimization techniques of clustering/blocking and sliding window, a new phase of similarity match optimization. The framework allows to customize a deduplication workflow using a configuration file and a rich set of available libraries for comparators and clustering functions. The record collection data model can be adapted to any specific context and T-match function allows for the definition of smart and efficient similarity functions, which may combine multiple and complementary similarity strategies. The implementation using Spark contributes to the computation optimization because of the parallelization of the tasks in the clustering and similarity-matching phase. T-match gains further execution time by anticipating the execution of *NO_MATCH* decisions and postponing time-consuming decisions, such as the $AuthorsMatch$ in the example. As proven by the reported experiments the hypothesis is not only intuitively correct but brings in some scenarios substantial performance gains. When used to analyze big data collections, time-saving is key for many reasons: the execution of experiments to improve a configuration, speeding up the generation of quality data in production systems or saving time that can be "spent" to improve the recall and precision by relaxing clustering and sliding window approaches, i.e., large numbers of blocks and increased window size.

On the other hand, time-saving depends on the ability to identify smart exit strategies applicable to a considerable percentage of the pair-wise comparisons. For example, if the publication record collection used for the experiments features correct and corresponding PIDs for all records, the $PublicationTreeMatch$ execution time would be further improved; on the contrary, if no PIDs are provided, the execution time would increase and get closer to the $PublicationWeightedMatch$. The two functions would perform identically if the records feature no differences in the title, making the $AuthorsMatch$ title determinant of the final decision.

Acknowledgments

The work in this paper has been funded by the projects OpenAIRE-Nexus (grant agreement ID 101017452) and FAIRCORE4EOSC (grant agreement ID 101057264).

References

[1] L. Li, Entity resolution in big data era: Challenges and applications, in: C. Liu, L. Zou, J. Li (Eds.), Database Systems for Advanced Applications, Springer International Publishing, Cham, 2018, pp. 114–117.

[2] M. Kejriwal, Entity resolution in a big data framework, in: B. Bonet, S. Koenig (Eds.), Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA, AAAI Press, 2015, pp. 4243–4244. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9294.

[3] E. Rahm, E. Peukert, Large scale entity resolution, in: S. Sakr, A. Y. Zomaya (Eds.), Encyclopedia of Big Data Technologies, Springer, 2019. URL: https://doi.org/10.1007/978-3-319-63962-8_4-1. doi:10.1007/978-3-319-63962-8_4-1.

[4] V. Christophides, V. Efthymiou, T. Palpanas, G. Papadakis, K. Stefanidis, End-to-end entity resolution for big data: A survey, 2019. arXiv:1905.06397.

[5] M. Nentwig, M. Hartung, A. N. Ngomo, E. Rahm, A survey of current link discovery frameworks, Semantic Web 8 (2017) 419–436. URL: https://doi.org/10.3233/SW-150210. doi:10.3233/SW-150210.

[6] J. a. Paulo, J. Pereira, A survey and classification of storage deduplication systems, ACM Comput. Surv. 47 (2014). URL: https://doi.org/10.1145/2611778. doi:10.1145/2611778.

[7] A. Venish, K. Sankar, Framework of data deduplication: A survey, Indian Journal of Science and Technology 8 (2015). doi:10.17485/ijst/2015/v8i26/80754.

[8] M. De Bonis, P. Manghi, C. Atzori, Fdup: a framework for general-purpose and efficient entity deduplication of record collections, PeerJ Computer Science 8 (2022) e1058.

[9] P. Manghi, C. Atzori, M. De Bonis, A. Bardi, Entity deduplication in big data graphs for scholarly communication, Data Technologies and Applications ahead-of-print (2020). doi:10.1108/DTA-09-2019-0163.

[10] M. De Bonis, C. Atzori, S. La Bruzzo, miconis/fdup: Fdup v4.1.10, 10.5281/zenodo.6011544, 2022. URL: https://doi.org/10.5281/zenodo.6011544. doi:10.5281/zenodo.6011544.

[11] M. De Bonis, 10mi openaire publications dump, 10.5281/zenodo.5347803, 2021. URL: https://doi.org/10.5281/zenodo.5347803. doi:10.5281/zenodo.5347803.

[12] P. Manghi, C. Atzori, A. Bardi, M. Baglioni, J. Schirrwagen, H. Dimitropoulos, S. La Bruzzo, I. Foufoulas, A. Löhden, A. Bäcker, A. Mannocci, M. Horst, P. Jacewicz, A. Czerniak, K. Kiatropoulou, A. Kokogiannaki, M. De Bonis, M. Artini, E. Ottonello, A. Lempesis, A. Ioannidis, N. Manola, P. Principe, Openaire research graph dump, 10.5281/zenodo.4707307, 2021. URL: https://doi.org/10.5281/zenodo.4707307. doi:10.5281/zenodo.4707307.