Parallel Meta-blocking: Realizing Scalable Entity Resolution over Large, Heterogeneous Data

Vasilis Efthymiou[1], George Papadakis[2], George Papastefanatos[3], Kostas Stefanidis[1], Themis Palpanas[4]
[1] *ICS-FORTH, Greece* {*vefthym, kstef*}*@ics.forth.gr*
[2] *University of Athens, Greece* *gpapadis@di.uoa.gr*
[3] *Athena Research Center, Greece* *gpapas@imis.athena-innovation.gr*
[4] *Paris Descartes University, France* *themis@mi.parisdescartes.fr*

***Abstract*—Entity resolution constitutes a crucial task for many applications, but has an inherently quadratic complexity. Typically, it scales to large volumes of data through blocking: similar entities are clustered into blocks so that it suffices to perform comparisons only within each block. Meta-blocking further increases efficiency by cleaning the overlapping blocks from unnecessary comparisons. However, even Meta-blocking can be time-consuming: applying it to blocks with 7.4 million entities and $2.2{\cdot}10^{11}$ comparisons takes almost 8 days on a modern high-end server. In this paper, we parallelize Meta-blocking based on MapReduce. We propose a simple strategy that explicitly creates the core concept of Meta-blocking, the blocking graph. We then describe an advanced strategy that creates the blocking graph implicitly, reducing the overhead of data exchange. We also introduce a load balancing algorithm that distributes the computationally intensive workload evenly among the available compute nodes. Our experimental analysis verifies the superiority of our advanced strategy and demonstrates an almost linear speedup for all meta-blocking techniques with respect to the number of available nodes.**

I. Introduction

Entity resolution (ER) is the task of mapping different entities to the same real-world object [6]. In the context of Big Web Data, it constitutes a batch process of quadratic complexity that is confronted with two Vs: *volume*, as it receives a large number of profiles as input, and *variety*, because the profiles are described by diverse schemata [16], [17] (velocity appears in Incremental ER). Volume is typically addressed by *blocking*, which places similar profiles into blocks and performs comparisons within each block. Variety is addressed by schema-agnostic blocking methods, which completely disregard attribute names; for instance, Token Blocking [16] creates one block for every token shared by at least two entities. Most of these blocking methods are *redundancy-positive*, placing profiles into multiple blocks so that the more blocks two profiles share, the more likely they are to be matching [16]. On the flip side, they entail two kinds of unnecessary comparisons: the *rendundant* ones repeatedly compare the same profiles in multiple blocks, while the *superfluous* ones involve non-matching profiles. For example, the blocks b_2 and b_4 in Figure 1(a) contain one redundant comparison each, repeated in b_1 and b_3; assuming that profiles e_1 and e_2 match with e_3 and e_4, b_5, b_6, b_7

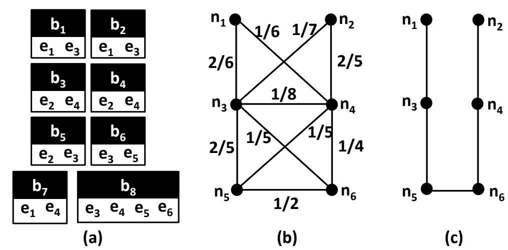

Figure 1. (a) A set of redundancy-positive blocks, (b) the corresponding blocking graph with Jaccard similarity as edge weight, and (c) one of the possible pruned blocking graphs.

and b_8 contain superfluous comparisons. In total, there are 13 comparisons: (4·3/2=)6 in b_8 and 1 in each of the 7 remaining blocks; 3 of these comparisons are redundant and 8 superfluous. Such comparisons increase the computational cost without contributing any identified duplicates.

Current state-of-the-art. *Block Processing* is the task of discarding unnecessary comparisons to enhance the efficiency of block collections. Established approaches include Iterative Blocking [19] and *Meta-blocking* [17], with the latter consistently outperforming the former in both effectiveness and time efficiency [17]. This is achieved as follows. First, it transforms the input block collection B into a *blocking graph* G_B that contains a node n_i for every profile e_i in B and an edge $<n_i, n_j>$ for every pair e_i and e_j that share at least one block. Every edge $<n_i, n_j>$ is associated with a weight $w_{i,j} \in [0, 1]$, analogous to the likelihood that the adjacent profiles are matching. For instance, the graph in Figure 1(b) is extracted from the blocks in Figure 1(a) using Jaccard similarity for weighting the edges. Note that there are no parallel edges, thus eliminating all redundant comparisons at no cost in recall (i.e., without missing any matching comparison). Second, the edges with low weights are discarded according to a pruning criterion so as to eliminate part of the superfluous comparisons. For instance, the graph G'_B in Figure 1(c) is derived from the graph in Figure 1(b) by discarding the edges with weight lower than the average one (1/4). G'_B is then transformed into a collection B' by creating a new block for every retained edge. G'_B yields 5 blocks, each containing a pair of entities. Out of the 5 comparisons, only 3 are superfluou, as the edges $<n_1, n_3>$ and $<n_2, n_4>$ connect matching entities. Compared to the initial block collection in Figure 1(a), the comparisons were reduced by 62% without missing any pair of duplicates.

Scalability Limitations. Theoretically, Meta-blocking involves a linear time complexity with respect to the number of comparisons in the input block collection [17]. In practice, though, its running time depends also on the average number of blocks associated with every profile. The reason is that the edge $<n_i, n_j>$ is weighted after estimating the intersection of the list of blocks associated with e_i and e_j. Therefore, the higher the redundancy in a block collection, the more time-consuming the processing of Meta-blocking.

So far, the largest dataset processed by Meta-blocking involves 3.4M profiles ($4.0 \cdot 10^{10}$ comparisons), each placed in 15 blocks on average ($D_{dbpedia}$ and DB_C in Tables I and II, respectively) [17]. Using a high-end server with Intel i7 (3.40GHz), 64 GB of RAM and Debian Linux 7, Meta-blocking requires 3 hours to execute. To assess its scalability, we tested it on 7.4M profiles ($2.2 \cdot 10^{11}$ comparisons), each associated with 40 blocks on average ($D_{freebase}$ and FR_D in Tables I and II, respectively). Using the same server, the required time raised to 186 hours (~8 days), i.e., a 2x increase in the size of the input resulted in a 62x increase in execution time. Therefore, even as a pre-processing step for ER, Meta-blocking is a heavy computational task with serious efficiency limitations at the scale of the Web. We expect this problem to aggravate over time, as Web Data grow constantly, both in terms of the number of entity profiles and the amount of information inside each profile (see `http://stats.lod2.eu`). To overcome these limitations of the existing serialized techniques, novel distributed approaches are required.

Proposed Solution. In this paper, we adopt MapReduce for parallelizing Meta-blocking and scaling its techniques to voluminous Web Data collections. We provide two strategies. The first one explicitly targets the blocking graph, building and storing all edges along with their weights. Although MapReduce leads to a significant speedup, it bears a high I/O cost that may become the bottleneck, when building very large graphs. The second approach overcomes this shortcoming, by using implicitly the blocking graph; it enriches the input block collection with the necessary information for computing the edges' weights on demand, without explicitly storing them. To avoid potential bottlenecks associated to the computation-intensive parts of our MapReduce functions, we also introduce a novel load balancing algorithm. It exploits the power law distribution of block sizes in redundancy-positive collections to split them in partitions of equivalent computational cost (i.e., total number of comparisons).

Finally, we provide an extensive experimental evaluation of our methods over the Hadoop environment (`https://hadoop.apache.org`). We apply the main Meta-blocking configurations to four large-scale, real-world datasets and measure the qualitative performance as well as the corresponding running times. The outcomes exhibit an almost linear speedup with respect to the available nodes. To facilitate other researchers to experiment with parallel Meta-blocking, we have publicly released the data and the implementation of our methods (See `https://github.com/vefthym/ParallelMetablocking`).

Contributions. In summary, these are our contributions:

- We provide a parallelized version of the Meta-blocking workflow based on the MapReduce paradigm. For each stage of the workflow, we offer two alternative strategies: a basic and an advanced one of higher scalability.
- We present a load balancing technique that deals with skewness in the input block collection, splitting it evenly into partitions with the same number of comparisons.
- We demonstrate the high performance and the linear speedup of our techniques through a thorough experimental evaluation over four real, voluminous datasets.

The rest of the paper is organized as follows: Section II describes related work, Section III provides the preliminaries for blocking and Meta-blocking, in Section IV we adapt Meta-blocking to MapReduce and evaluate its performance in Section V. We summarize our findings in Section VI.

II. Related Work

ER constitutes a well-studied problem [7], [6], [9]. Due to its quadratic complexity, a bulk of work aims at improving its scalability. *Parallel ER methods*, e.g., [5], [13], [10], exploit the processing power of multiple cores to minimize the ER response time. Recent approaches are based on MapReduce, which offers fault-tolerant, optimized execution for applications, distributed across independent nodes. Its programs consist of two consecutive procedures grouped together into *jobs*: `Map` receives a (`key`, `value`) pair and transforms it into one or more new pairs; `Reduce` receives a set of pairs that share the same `key` and are sorted according to their `value`, and performs a summary operation on them to produce a new, usually smaller set of pairs. Based on MapReduce, [3] introduces an approach in which a decision about matching two entity profiles triggers further decisions about matching their associated profiles. Similar approaches are used by other iterative techniques, which employ some partial results of the ER process to locate new matches (e.g., [1], [2], [8], [12]).

In a different line of work, *approximate techniques* aim to achieve a good balance between the number of identified duplicates and the number of executed comparisons. The most prominent technique is *blocking* [4]; it represents profiles by sets of keys and groups similar profiles into blocks based on similar or identical keys. Comparisons are then executed inside the resulting blocks. More recent methods, e.g., [16], target data of low structuredness, such as those stemming from the Web. Most of these techniques are redundancy-positive, yielding a large number of unnecessary comparisons when applied to large datasets. Yet, their blocks can be significantly enhanced by Meta-blocking.

This work bridges the gap between the two lines of research for ER over Web data, parallelizing Meta-blocking to achieve even higher scalability. A similar effort for tabular data is made in [14], [15], which adapt Standard Blocking and Sorted Neighborhood, respectively, to MapReduce.

III. Preliminaries

Blocking. An *entity profile* comprises a set of name-value pairs, uniquely identified through a global id; e_i denotes a profile with id i. Two profiles that refer to the same object are called *duplicates* or *matches*. A set of profiles is called *entity collection* (E); $D(E)$ stands for the duplicate profiles in E, and $|D(E)|$ for the number of duplicates in E. A blocking method groups the entities of a collection into clusters, called *blocks*; b_i is a block with id i. The number of entities in b_i is called *block size* ($|b_i|$), while the number of comparisons it involves is called *block cardinality* ($||b_i||$). Collectively, a set of blocks is called *block collection* (B); $|B|$ stands for its size, and $||B||$ for its *total cardinality*, which is the number of comparisons it involves, i.e., $||B|| = \sum_{b_i \in B} ||b_i||$. The set of blocks containing a particular entity e_i is denoted by $B_i(\subseteq B)$, with $|B_i|$ representing its size. Two entities e_i, e_j placed in the same block are called *co-occurring*, and their comparison, $c_{i,j}$, is called *matching* if e_i, e_j are duplicates; $B_{i,j}$ represents the set of blocks they co-occur in and its size, $|B_{i,j}|$, stands for the number of blocks they share.

Typically, the performance of a block collection is independent of the entity matching method that executes the pair-wise comparisons [4], [17]. The main assumption is that two duplicates are detected as long as they *co-occur* in at least one block. The set of co-occurring duplicate entities is denoted by $D(B)$, with $|D(B)|$ representing its size. In this context, two established measures are used for assessing the performance of a block collection [4], [16]:

• *Pairs Completeness* (PC) is analogous to recall, estimating the portion of existing pairs of duplicates that are co-occurring: $PC = |D(B)|/|D(E)|$. It is defined in the interval $[0, 1]$, with higher values indicating better recall.

• *Pairs Quality* (PQ) is analogous to precision, estimating the portion of executed comparisons that involve a non-redundant pair of duplicates: $PQ = |D(B)|/||B||$. Defined in the interval $[0, 1]$, higher values indicate better precision.

Ideally, the goal of blocking is to maximize both PC and PQ. However, there is a trade-off between these measures: the more comparisons are contained in B, the more duplicates are co-occurring and the higher PC gets. Given, though, that $||B||$ increases quadratically for a linear increase in $|D(B)|$ [11], PQ is reduced. For this reason, the goal of blocking methods in practice is to achieve a good balance between the two measures – with an emphasis on recall.

Meta-blocking. Redundancy-positive blocking trades high PC for low PQ, i.e., it yields a large number of unnecessary comparisons to achieve high recall. Meta-blocking operates on its blocks to tip the balance in favor of precision at a small cost in recall. It restructures a redundancy-positive block collection B into a new one B' such that $PC(B') \approx PC(B)$ and $PQ(B') \gg PQ(B)$ [17]. Its performance depends on two parameters that affect the pruning of the blocking graph: the weighting and the pruning scheme.

Common Blocks Scheme	$CBS(e_i, e_j, B) = \lvert B_{ij} \rvert$
Enhanced Common Blocks Scheme	$ECBS(e_i, e_j, B) = CBS(e_i, e_j, B) \cdot \log\frac{\lvert B\rvert}{\lvert B_i\rvert} \cdot \log\frac{\lvert B\rvert}{\lvert B_j\rvert}$
Jaccard Scheme	$JS(e_i, e_j, B) = \frac{\lvert B_{ij}\rvert}{\lvert B_i\rvert + \lvert B_j\rvert - \lvert B_{ij}\rvert}$

Figure 2. The formal definition of the weighting schemes.

The *weighting scheme* receives as input the entities defining an edge in G_B along with the block collection B and estimates the corresponding weight. We focus on three schemes [17], formally defined in Figure 2. CBS captures the fundamental property that the more blocks two entities share, the more likely they are matching. ECBS improves CBS by discounting the contribution of entities participating in many blocks. Finally, JS estimates the portion of blocks shared by two entities. In all cases, their weights are restricted to $[0, 1]$ through normalization.

For the pruning scheme, there are four options [17]:

• *Weighted Edge Pruning* (WEP) retains all edges with a weight higher than the overall mean one.

• *Cardinality Edge Pruning* (CEP) retains the top-K edges of the entire blocking graph, where $K = \lfloor \sum_{b_i \in B} |b_i|/2 \rfloor$.

• *Weighted Node Pruning* (WNP) amounts to the average edge weight of each neighborhood.

• *Cardinality Node Pruning* (CNP) retains, for each neighborhood, the top-k edges with $k = \lfloor \sum_{b_i \in B} |b_i|/|E| - 1 \rfloor$.

In this work, we adapt to the MapReduce framework, the combination of all pruning schemes with the CBS, ECBS and JS weighting schemes[1].

All pruning schemes benefit greatly from *Block Filtering* [18], which like Meta-blocking cleans a block collection from many unnecessary comparisons. Instead of using a graph, though, it simply removes every entity from the least important of its blocks. The main assumption is that the larger a block is, the less important it is for its entities. In more detail, Block Filtering orders the blocks of B in ascending order of cardinality and retains every entity e_i in the top N_i blocks of B_i (i.e., the N_i smallest blocks that contain e_i), where $N_i = \lfloor r \times |B_i| \rfloor$ and $r \in [0, 1]$ is the *ratio* of Block Filtering. In this work, we employ Block Filtering as an integral part of our parallelized approach, setting $r = 0.80$. This value was experimentally verified to prune at least 50% of the blocking graph's edges, while having a negligible impact on recall [18].

[1] Due to lack of space, we cover the remaining weighting schemes, namely ARCS and EJS, in the extended version of our paper: http://www.csd.uoc.gr/~vefthym/MetaBlockingExt.pdf.

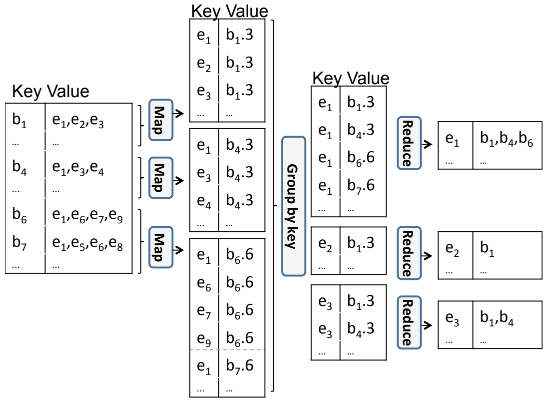

Figure 3. An example for the advanced strategy of Block Filtering.

IV. Approach

We now elaborate on the adaptation of Meta-blocking to MapReduce. The serialized workflow consists of two consecutive stages: the first one applies Block Filtering to the input block collection B, while the second one applies Meta-blocking to yield the final, restructured collection B'. The parallelized counterpart consists of three stages. Again, the first one applies Block Filtering to the input block collection and the last one Meta-blocking. The only difference is in the second stage, which preprocesses the blocks to transform them into a suitable form for Parallel Meta-blocking.

For every stage of the parallelized workflow, we consider two different approaches: *(i)* a basic strategy, which applies a straightforward adaptation with high I/O between the nodes, and *(ii)* an advanced strategy, which reduces the overhead of data exchange through a more elaborate processing. We analyse each stage separately, in Sections IV-A to IV-C. In Section IV-D, we introduce a novel algorithm for load balancing that applies to both strategies. In all cases, special care was taken to minimize the I/O between the independent nodes. Part of this effort focused on optimizing our representation model. Instead of using the textual blocking keys and URIs to identify the blocks and the entities, respectively, our model relies exclusively on numbers: blocks and entities are uniquely identified by integer ids; edges are represented by the concatenation of the adjacent entity ids.

A. Stage 1: Block Filtering

The first stage applies Block Filtering to the input blocks in order to reduce the size of the blocking graph. Central to this procedure is the sorting of blocks in ascending order of cardinality, from the smallest to the largest one. Depending on how this sorting is performed, we present two possible approaches for adapting Block Filtering to MapReduce.

The basic strategy orders once and globally all input blocks, using two MapReduce jobs that exploit the automatic sorting of the input to the `reduce` function. The advanced strategy employs a single MapReduce job that orders locally the blocks associated with every entity at the cost of repeating some computations across the independent nodes.

For both strategies, every `(key, value)` pair of the input corresponds to a block b_k; the key is the block id, while the value contains the list of the entity ids placed in b_k: `key`$=k$ and `value`$=\{i, j, \dots, m\}$ for $b_k=\{e_i, e_j, \dots, e_m\}$. The output of both strategies comprises the N most important blocks associated with the individual entities. Every key denotes the id of an entity e_i, while the corresponding value contains the list of block ids still containing e_i: `key`$=i$ and `value`$=B'_i$.

1) Basic Strategy: It employs two jobs. The first one sorts all blocks globally in ascending order of cardinality, producing a sorted list B_{sorted}. Specifically, the `map` function receives a block id k along with the entities contained in b_k. It computes the corresponding cardinality, $||b_k||$, and emits a ($||b_k||$, k) pair. All pairs are sorted in descending order of their keys (i.e., cardinalities), before they are forwarded as input to the single `reduce` function. The reducer extracts and stores to the disk the values of the sorted input, i.e., the block ids that form B_{sorted}. The second job uses B_{sorted} to identify the most important blocks for each entity. The `map` function gets the same input as the first job: a block id along with the entity ids it contains. For every entity e_i contained in the given block b_k, it emits a pair (i, k). All pairs having the same key are grouped together so that the `reduce` function receives as input all block ids assigned to a specific entity e_i (i.e., `key`$=i$, `value`$=B_i$). It loads from the disk the sorted list of block ids, B_{sorted}, and uses it to get the ranking position of every block. The N blocks with the highest ranking positions form the list of retained block ids B'_i, which are emitted as output: `key`$=i$, `value`$=B'_i$.

2) Advanced Strategy: It uses a single job that provides the `reduce` function with the necessary information for sorting the blocks of each entity locally. The `map` function gets as input the id and the entities of a block b_k and estimates its cardinality, $||b_k||$. For every entity $e_i \in b_k$, it emits a pair with the entity id as the key, while the (composite) value concatenates the id and the cardinality of block b_k: `key`$=i$ and `value`$=k.||b_k||$. The `reduce` function gathers all blocks associated with an entity e_i along with their cardinality. It sorts them in ascending number of comparisons and extracts the top N elements from the resulting list to form B'_i. Similar to the basic strategy, it then emits a pair (i, B'_i).

Figure 3 illustrates the functionality of the advanced strategy of Block Filtering. For the three entities e_1, e_2 and e_3 of b_1, we emit in the `Map` phase a pair with each of them as the `key` and $b_1.3$ as `value`, since there are three comparisons in this block. In the `Reduce` phase, we gather all four pairs having e_1 as key and keep only the top-3 blocks for this entity. Thus, we discard b_7 from the blocks of e_1.

B. Stage 2: Preprocessing

This stage prepares the data that will be processed by the pruning algorithm in the third stage. Its output actually determines the complexity of the pruning algorithm: the more computations are performed by Preprocessing and are integrated into its output, the simpler is the functionality of the pruning algorithms and vice versa. This trade-off gives rise to two different strategies, which share the same input (i.e., the outcome of Block Filtering), but differ in their output. Basic Preprocessing explicitly creates the blocking graph: it performs all weight computations and stores all edges to the disk in order to simplify the functionality of the pruning algorithm. Advanced Preprocessing defers all weight computations, but facilitates them by enriching the input of the pruning algorithms with all the necessary information. Again, the basic strategy involves two jobs, while the advanced one uses just one job.

1) Basic Strategy: The first job transforms the output of Block Filtering into a block collection. Its `map` function receives as key the id of an entity e_i and as value the list of associated blocks, B_i. It swaps values and keys, emitting for every block $b_k \in B_i$ a pair $(k, i.|B_i|)$, where k and i are the block and the entity id, respectively, while $|B_i|$ is the number of blocks containing e_i after Block Filtering. The `reduce` function groups together all entities contained in a block b_k and reproduces all comparisons. For every comparison $c_{i,j}$ between entities e_i and e_j, it emits the concatenation of their ids as key (`key`=$i.j$) and $|B_i|.|B_j|$ as value – this information is necessary for the ECBS and JS weighting schemes.

The second job consists of an identity mapper and a `reduce` function that estimates the weight for every edge of the blocking graph. The value list of its input, V, clusters together the information pertaining to the edge $<n_i, n_j>$ identified by the input key. Based on them, the reducer computes the corresponding edge weight $w_{i,j}$ from the formulas in Figure 2. For example, we simply have $w_{i,j} = |V|$ for CBS, as the size of the value list equals the number of common blocks, $|B_{i,j}|$. As output, the reducer emits a pair with the id and the weight of the edge: `key`=$i.j$ and `value`=$w_{i,j}$.

2) Advanced Strategy: The key to this approach is that every edge $<n_i, n_j>$ of the blocking graph G_B corresponds to a non-redundant comparison $c_{i,j}$ in the block collection B. A comparison $c_{i,j}$ in b_k is *non-redundant* only if it satisfies the Least Common Block Index (LeCoBI) condition. That is, if the id of b_k equals the least common block id of the entities e_i and e_j: $k{=}min(B_i \cap B_j)$ [16]. To assess the LeCoBI condition for two entities e_i and e_j, we need to compare the lists of associated blocks, B_i and B_j (for higher efficiency, their elements should be sorted in ascending order of block ids). Advanced Preprocessing integrates this information to its output, so that all edge and weight computations are carried out by the pruning algorithm.

This functionality is performed by one job. The `map` function receives as input the id of an entity e_i as key and the associated blocks B_i as values. First, it sorts B_i in ascending order of block ids. Then, for every block $b_k \in B_i$, it emits its id as the key, while the value concatenates the id of e_i with the entire sorted list B_i: `key`=k, `value`=$i.B_i$. MapReduce then reassembles all blocks, by grouping together all pairs with the same key. The `reduce` function receives as input the entity list of a specific block along with the blocks that are associated with every individual entity and emits the same `(key, value)` pairs as output: `key`=k and `value`=$\{i.B_i, j.B_j, \ldots, m.B_m\}$.

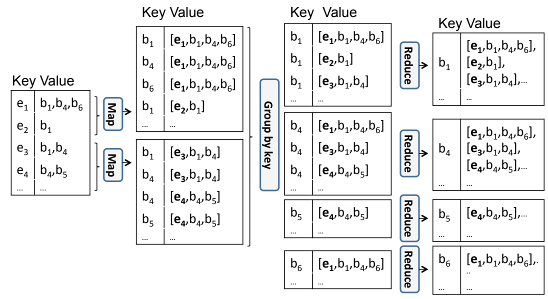

Figure 4. An example for the advanced strategy of Preprocessing.

Figure 4 provides an example for the functionality of the advanced strategy of Preprocessing. For each block b_1, b_4 and b_6, to which e_1 belongs, we emit a pair with their block ids as `key` and e_1, concatenated with b_1, b_4, b_6 as `value` in the `Map` phase. In the `Reduce` phase, all the entities of b_1 are grouped together (i.e., e_1, e_2 and e_3), each accompanied with the block ids in which it belongs. We just concatenate them and emit them as the `value` of the `key`=b_1.

C. Stage 3: Meta-blocking

This stage applies one pruning algorithm to the output of Preprocessing and yields a set of retained edges; every edge corresponds to a new block that is part of the final, restructured block collection. We present two strategies for each algorithm: the basic and the advanced one, applied on top of the basic and the advanced preprocessing outputs.

1) Weighted Edge Pruning: Both strategies employ a single job that estimates the average edge weight in the `Map` phase and discards the edges that do not exceed it in the `Reduce` phase. They use the same `reduce` function and differ only in the `map` function.

Basic Strategy. This identity mapper receives the id of an individual edge $<n_i, n_j>$ as key (`key`=$i.j$) and its weight $w_{i,j}$ as value. Before forwarding the input to the reducer, it updates two counters used for estimating the average edge weight: the size of the blocking graph $|E_G|$ and the total edge weight tw. The reducer receives the id and the weight of an individual edge. If the weight is greater than the mean weight of the graph ($w_{i,j} > tw/|E_G|$), the input edge is retained and, thus, the input pair ($i.j$, $w_{i,j}$) is emitted as output.

Advanced Strategy. It operates on the enriched description of an individual block b_k: the input key contains its id (k), while the input value contains a list with the entity ids in b_k and the blocks ids associated with each entity, i.e., `value` = $\{i.B_i, j.B_j, \ldots\}$. The `map` function iterates over all comparisons in b_k and assesses the LeCoBI condition for the involved entities. For every non-redundant comparison $c_{i,j}$, it estimates the corresponding edge weight $w_{i,j}$ and emits it along with the edge id: `key`=$i.j$ and `value`=$w_{i,j}$. It also

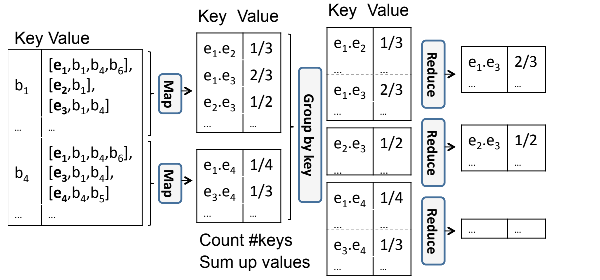

Figure 5. An example for the advanced strategy of Weighted Edge Pruning, using the JS weighting scheme.

updates the two counters that are used in the reduce phase, $|E_G|$ and tw. As in the Basic strategy, the reducer emits the input pairs $(i.j,\ w_{i,j})$ for which $(w_{i,j} > tw/|E_G|)$.

In Figure 5, we present the functionality of the advanced strategy of `WEP`, with an example using the `JS` weighting scheme. Applying the `map` function to block b_1, which contains the entity profiles e_1, e_2 and e_3, we output, for every non-redundant comparison the id of the comparison as `key` (i.e., $e_1.e_2$, $e_1.e_3$ and $e_2.e_3$) and the weight of the comparisons as `value`. According to the *JS* weighting scheme, e_1-e_2 weight is 1/3, because the e_1, e_2 entities share only one block (b_1) from all 3 blocks they belong to. Assuming that the average weight is 1/3, in the `reduce` function we emit only the pairs with a weight above 1/3, so we prune the comparisons e_1-e_2, e_1-e_4 and e_3-e_4.

2) Cardinality Edge Pruning: Ideally, we could gather all edges in a single node and sort them in descending weight to retain the top K ones. In practice, though, this approach does not scale to large blocking graphs with millions of nodes and billions of edges, due to limited memory resources. To overcome this limitation, we convert the global cardinality threshold into a global weight threshold. We use one job to compute the minimum edge weight w_{min} such that at least K edges have a weight greater than or equal to it. Then, a second job outputs exactly K edges with a weight greater than or equal to w_{min}. For every job, the two strategies again differ in the mapper, but share the same reducer.

Basic Strategy. For both jobs, the mapper receives as input key the id of an individual edge $<n_i, n_j>$ $(i.j)$ and as input value the corresponding weight $(w_{i,j})$. The `map` function of the first job emits a pair $(w_{i,j},\ 1)$, enabling the reducer to compute the minimum edge weight w_{min}. The reducer receives as input the list of all distinct weights, sorted in descending order, along with their frequencies (i.e., the number of edges with the same weight). It iterates this list starting from the largest weight and keeps a counter with the number of edges that have a weight greater than or equal to the current one. As soon as the counter reaches K, the reducer stops and stores the current weight w_{min} to the disk.

The `map` function of the second job processes the same input $<n_i, n_j>$ $(i.j)$, $(w_{i,j})$, swaps keys with values and emits a pair $(w_{i,j},\ i.j)$, only if $w_{i,j} \geq w_{min}$. It is possible that the overall number of edges $<n_i, n_j>$, with $w_{i,j} \geq w_{min}$, is

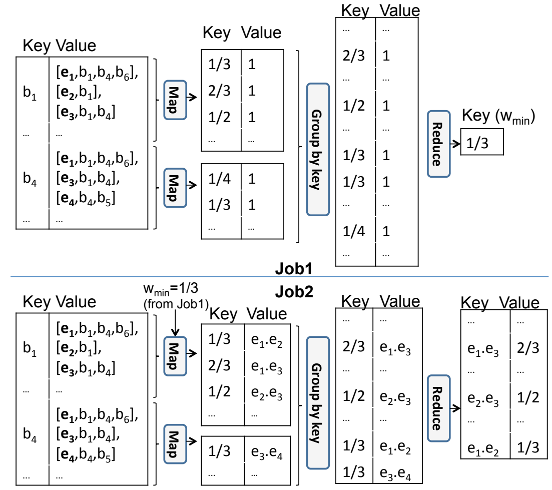

Figure 6. An example for the advanced strategy of Cardinality Edge Pruning, using the JS weighting scheme.

larger than K, due to ties. The single reducer of the second job addresses this issue, ensuring that exactly K edges are retained. It receives as input all pairs of edge weights and ids $(w_{i,j}, i.j)$, for which $w_{i,j} \geq w_{min}$. They are automatically sorted in descending order, from the largest weight to the lowest. The reducer extracts the top K elements and emits them, after swapping keys and values.

Advanced Strategy. For both jobs, the mapper iterates over all comparisons of the input block, using its enriched description. For every non-redundant comparison $c_{i,j}$, it computes the edge weight $w_{i,j}$ from the block ids associated with e_i and e_j. Then, the `map` function of the first job emits a pair $(w_{i,j},\ 1)$, while the `map` function of the second job emits a pair $(w_{i,j},\ i.j)$, only if $w_{i,j} \geq w_{min}$. The `reduce` function for both jobs is the same with that of the Basic Strategy.

In Figure 6, we provide an example with the functionality of the advanced strategy of `CEP`, using the same input with Figure 5 and the `JS` weighting scheme. In the `map` function of the first job, we process each block and emit the weights of its non-redundant comparisons as `keys` (e.g., for b_1 the non-redundant comparinsons are $e_1.e_2$, $e_1.e_3$ and $e_2.e_3$), and 1 as `value`. In the `reduce` function, we retrieve, in descending order of weight, the first k input pairs and emit the current `key` as w_{min}; in our example we assume to be 1/3. In the second job, we use the same `map` function, this time emitting the comparison ids as `values`, for those comparisons whose weight is greater than or equal to $w_{min} = 1/3$. Hence, we prune the comparison e_1-e_4 already from the `map` phase. In the `reduce` function, we emit each input pair, sorted in descending order of weight, until we have emitted the k-th pair, which is e_1-e_2 (pairs with the same weight are sorted randomly).

3) Weighted & Cardinality Node Pruning: Both node pruning schemes use one job with the same `map` function. They differ in the `reduce` function, which applies their pruning logic to an individual node neighborhood.

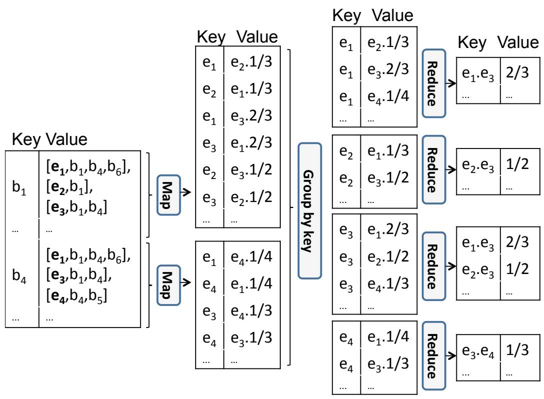

Figure 7. An example for the advanced strategy of Weighted Node Pruning, using the JS weighting scheme.

Basic Strategy. The `map` function takes as input key the id of an individual edge <n_i, n_j> ($i.j$) and as input value, the corresponding weight ($w_{i,j}$). To ensure that each reducer gathers all edges adjacent to a specific node, it emits two `(key, value)` pairs – one for each of the adjacent entities. In each case, the key contains one of the entity ids (i or j), while the value concatenates the other entity id with the edge weight ($j.w_{i,j}$ or $i.w_{i,j}$).

WNP Reduce Function. Its input key comprises the id i of an entity e_i that defines a neighborhood $N_G(v_i)$ in the blocking graph G. Its input value comprises the adjacent node/entity ids concatenated with the respective edge weights. From them, it estimates the average weight of the neighborhood, $\bar{w}_i$. Then, it iterates over all adjacent edges and discards those that are assigned a lower weight. For each retained edge <n_i, n_j>, it emits a pair ($i.j$, $w_{i,j}$).

CNP Reduce Function. It receives the same input as the WNP `reduce` function and orders all edges of the neighborhood in descending order of weight. For the top k ones, it emits their id with their weight – a pair ($i.j$, $w_{i,j}$).

Advanced Strategy. The `map` function of the advanced strategy operates on the enriched description of an individual block, iterating over all its comparisons. For every non-redundant comparison, it computes the corresponding edge weight from the associated block ids and emits two `(key, value)` pairs, one for each of the adjacent entities – just like the basic `map` function. The aforementioned WNP and CNP `Reduce` functions process the input.

Figure 7 shows an example of the advanced strategy of `WNP` to the same input as that of Figures 5, 6, using `JS`. In the `map` function, for the comparison e_1-e_2, we emit the pairs $(e_1, e_2.1/3)$ and $(e_2, e_1.1/3)$. In the `Reduce` phase, we group all the pairs with `key`=e_1 and calculate, for this group, a local weight threshold (e.g., 1/3). Then, for the group of e_1, we emit only the pairs with a weight higher than 1/3, i.e., e_1-e_3, which has a weight of 2/3. Accordingly, the advanced strategy of `CNP` differs only in the last step, in which we emit only the top-k pairs of each group. If we assume that we want the top-2 comparisons that involve e_1, out of the three comparisons shown in the group of e_1, we would emit $(e1.e3, 2/3)$ and $(e1.e2, 1/3)$.

Algorithm 1: Load Balancing.

```
   Input: B the current block collection
   Output: P the set of block partitions
1  B′ ← sort(B);      // sort in descending cardinality
2  b0 ← B′.remove(0); // remove largest block
3  maxCost ← ||b0||;  // max comparisons per partition
4  P0 ← {b0};        // first partition
5  Q ← {P0};     // priority queue, sorting partitions in ascending
   cost
6  while B′ ≠ {} do           // while not empty
7      bi ← B′.remove(0);  // remove first block
8      Phead ← Q.poll();    // get lowest cost partition
9      totalCost ← ||bi|| + Phead.currentCost();
10     if totalCost ≤ maxCost then
11         Phead ← Phead ∪ {bi};  // add to partition
12     else
13         Pi ← {bi};    // create new partition
14         Q.add(Pi);  // add to queue
15     Q.add(Phead);  // place back to queue
16 return Q;
```

D. Load Balancing

The default load balancing of Hadoop creates a predetermined number of partitions using a hash function. It involves a negligible overhead and is expected to work fine for jobs that entail linear processing of the input data. This applies to most of the functions defined above, but they are expected to account for a small portion of the overall computational cost. The main computational cost pertains to functions with quadratic complexity: they receive an individual block and iterate over all the comparisons it contains, estimating (part of) the corresponding edge weights. These are the `reduce` function in the first job of the basic strategy for Stage 2 and all `map` functions of the advanced strategies for Stage 3. For these functions, the default partitioning disregards the block cardinalities and leads to significant skews in the workload distribution (see Section V).

To address this issue, we developed a specialized algorithm for load balancing. Our goal is to split the input blocks into partitions that share almost the same number of comparisons. The key idea is to exploit the power law distribution of block cardinalities that appears in redundancy-positive block collections: the vast majority of blocks involves one or two comparisons and the frequency of blocks decreases with larger cardinalities [16], [17]. For better results, the number of partitions is determined dynamically.

The functionality of our approach is outlined in Algorithm 1. It sorts the block collection in descending cardinality (Line 1) and removes the first and largest block, b_0, from it (Line 2). The maximum computational cost of each partition, *maxCost*, is set equal to the cardinality of b_0 (Line 3). A partition is created for b_0 (Line 4) and placed in the priority queue Q, which sorts partitions in ascending order of comparisons (Line 5); this means that the head of Q always corresponds to the partition with the smallest computational

	$D_{dbpedia}$		$D_{freebase}$	
	D_1	D_2	D_1	D_2
Entities $\lvert E \rvert$	1,190,733	2,164,040	3,157,726	4,204,942
Triples	$1.69 \cdot 10^7$	$3.50 \cdot 10^7$	$1.42 \cdot 10^8$	$3.90 \cdot 10^7$
Attribute Names	30,757	52,554	37,825	11,108
Triples per Entity	14.19	16.18	44.84	9.29
Duplicates $\lvert D(E) \rvert$	892,579		1,347,266	
BF Comparisons	$2.58 \cdot 10^{12}$		$1.33 \cdot 10^{13}$	

Table I
THE HETEROGENEOUS ENTITY COLLECTIONS WE EMPLOYED IN OUR EXPERIMENTS.

	$D_{dbpedia}$		$D_{freebase}$	
	DB_C	DB_D	FR_C	FR_D
Task	Clean-Clean ER	Dirty ER	Clean-Clean ER	Dirty ER
$\lvert B \rvert$	1,239,424	1,499,534	1,309,145	4,522,222
$\lVert B \rVert$	$4.23 \cdot 10^{10}$	$8.00 \cdot 10^{10}$	$1.05 \cdot 10^{11}$	$2.19 \cdot 10^{11}$
$\lvert D(B) \rvert$	891,708	891,572	1,319,050	1,271,512
BPE	15.30-16.08	14.79	75.55-4.43	40.12
PC	0.999	0.999	0.979	0.944
PQ	$2.11 \cdot 10^{-5}$	$1.12 \cdot 10^{-5}$	$1.26 \cdot 10^{-5}$	$5.82 \cdot 10^{-6}$

Table II
THE BLOCK COLLECTIONS THAT WERE USED AS INPUT TO META-BLOCKING.

cost so far. Subsequently, our algorithm iterates over the remaining blocks and examines whether the current block fits into the partition at the head of the queue, P_{head} (Lines 6-10); that is, it checks whether their combined cardinality is lower than *maxCost*. If so, the current block is added to P_{head} (Line 11); otherwise, it is placed in a new partition that is added to the queue (Lines 13-14). Then, P_{head} is placed again in Q (Line 15). The time complexity of our approach is dominated by the sorting of blocks, thus having a complexity of $O(|B| \log |B|)$. This means that our approach scales well to large block collections, involving a low overhead.

V. EXPERIMENTS

Setup. All approaches were implemented in Java v7, using Apache Hadoop v1.2.0 on a cluster[2] with 15 Ubuntu 12.04.3 LTS servers, one master and 14 slaves, each having 8 AMD 2.1 GHz CPUs and 8 GB of RAM. Each node can run 4 map or reduce tasks simultaneously, assigning 1024 MB to each task. The available disk space (4 TB) was equally partitioned to the nodes. Each time measurement was repeated twice and the average value was considered to eliminate external factors effects (e.g., network overhead). For Load Balancing, we employed the default mechanism of Hadoop for `map` and `reduce` functions with linear complexity; for those with a quadratic complexity, we partitioned the relevant blocks to the available nodes using Algorithm 1.

Datasets. We employ the largest datasets that have been applied to Meta-blocking. Their characteristics are presented in Table I. $D_{dbpedia}$ involves entities from two snapshots of DBpedia[3] infoboxes, which chronologically differ by 2 years – v3.0rc for D_1 and v3.4 for D_2. In total, there are 3.3M entities, of which less than 900,000 are common, having the same URL. This dataset has been employed widely in the literature [16], [17]. $D_{freebase}$ contains entities from the Billion Triple Challenge 2012[4]. In this case, D_1 encompasses entities from DBpedia and D_2 entities from Freebase[5]. To avoid noisy profiles, we disregard URIs that appear in just one triple. 7.4M entities were left, of which 1.3M are common according to the *owl:sameAs* statements.

Both datasets are suitable for *Clean-Clean ER*, where the goal is to identify the matching entities shared by two duplicate-free collections D_1 and D_2. We use them for *Dirty ER*, as well, merging D_1 and D_2 into a single collection with duplicates in itself; the goal is to partition the resulting collection into clusters of matching entities. To derive redundancy-positive block collections, we used Token Blocking and Block Purging [16], which simply discards the blocks that contain more than half the input entities. The characteristics of the resulting blocks are presented in Table II. In total, we have four block collections, two for each ER task, that vary significantly in their characteristics.

Measures. To assess the quality of the restructured block collections, we employ the established measures of Pairs Completeness PC (recall) and Pairs Quality PQ (precision) – see Section III. To assess the time efficiency of the Meta-blocking workflow, we use the *Overhead Time* ($OTime$). This is the time (in minutes) that intervenes between receiving a redundancy-positive block collection as input and returning the restructured blocks as output. The lower its value is, the more efficient is the corresponding workflow.

Load Balancing. We compare the performance of Algorithm 1 to the default Hadoop balancer through the distribution of partition cardinalities they produce (i.e., the total number of comparisons in the blocks of each partition). We summarize these distributions using minimum, maximum, median and mean partition cardinalities. The closer these measures are to each other, the more balanced the workload of each node. We applied both approaches to the input of Stage 2 and present the experimental outcomes in Table III.

Algorithm 1 produces a set of partitions with a practically constant distribution of cardinalities across all datasets. The four measures have identical values, while the standard deviation of the distribution is lower than 1. This means that the partitions differ by a handful of comparisons in the worst case. In contrast, the default balancer yields distributions with much larger variance. The standard deviation is an order of magnitude lower than the other measures for DB_C and DB_D, while for FR_C and FR_D it is almost equal to the average cardinality. The difference between the minimum and the maximum cardinality raises to four and one orders of magnitude for FR_C and FR_D. Thus, serious bottlenecks are expected to rise in all datasets. For this reason, we did not try to measure the actual running time of the default balancer. On the whole, we conclude that Algorithm 1 outperforms the default balancer of Hadoop. The only advantage of the latter is its low overhead, as it is integrated and optimized for Hadoop. However, our approach is quite scalable and

[2] ~okeanos (`https://okeanos.grnet.gr`) GRNET cloud service
[3] `http://dbpedia.org`
[4] `https://km.aifb.kit.edu/projects/btc-2012`
[5] `https://www.freebase.com`

	DB_C		DB_D		FR_C		FR_D	
	Algorithm 1	Default	Algorithm 1	Default	Algorithm 1	Default	Algorithm 1	Default
Partitions	442	223	378	223	2,042	1,674	1,735	1,119
Mininimum Part. Cardinality	$2.71 \cdot 10^7$	$4.18 \cdot 10^7$	$5.74 \cdot 10^7$	$7.88 \cdot 10^7$	$1.45 \cdot 10^7$	$3.21 \cdot 10^4$	$3.76 \cdot 10^7$	$9.81 \cdot 10^6$
Maximum Part. Cardinality	$2.71 \cdot 10^7$	$8.20 \cdot 10^7$	$5.74 \cdot 10^7$	$1.48 \cdot 10^8$	$1.45 \cdot 10^7$	$2.35 \cdot 10^8$	$3.76 \cdot 10^7$	$9.81 \cdot 10^7$
Median Part. Cardinality	$2.71 \cdot 10^7$	$5.83 \cdot 10^7$	$5.74 \cdot 10^7$	$9.59 \cdot 10^7$	$1.45 \cdot 10^7$	$2.02 \cdot 10^7$	$3.76 \cdot 10^7$	$9.01 \cdot 10^7$
Average Part. Cardinality	$2.71 \cdot 10^7$	$5.87 \cdot 10^7$	$5.74 \cdot 10^7$	$9.68 \cdot 10^7$	$1.45 \cdot 10^7$	$3.14 \cdot 10^7$	$3.76 \cdot 10^7$	$5.84 \cdot 10^7$
St. Dev. Part. Cardinality	**0.48**	**$7.59 \cdot 10^6$**	**0.19**	**$9.98 \cdot 10^6$**	**0.48**	**$3.01 \cdot 10^7$**	**0.27**	**$4.64 \cdot 10^7$**

Table III

THE DISTRIBUTION OF PARTITION CARDINALITIES PRODUCED BY ALGORITHM 1 AND THE DEFAULT LOAD BALANCER OF HADOOP.

		DB_C		DB_D		FR_C	FR_D
		Basic	Advan.	Basic	Advan.	Advan.	Advan.
Block Filtering		2	2	2	2	3	6
CEP	CBS	222	22	250	29	184	584
	ECBS	240	38	278	55	235	721
	JS	223	28	279	39	197	606
CNP	CBS	491	301	559	527	1,488	2,514
	ECBS	555	383	639	633	1,949	3,058
	JS	534	363	618	620	1,637	2,546
WEP	CBS	220	38	250	63	271	479
	ECBS	219	46	254	103	371	559
	JS	219	41	254	90	337	524
WNP	CBS	498	304	569	539	1,534	2,671
	ECBS	568	389	658	647	1,971	3,046
	JS	553	373	641	644	1,636	2,790

Table IV

OVERHEAD TIME IN MINUTES FOR ALL META-BLOCKING TECHNIQUES.

	$D_{dbpedia}$		$D_{freebase}$	
	DB_C	DB_D	FR_C	FR_D
$\|B\|$	1,239,315	1,499,422	1,308,970	4,521,129
$\|\|B\|\|$	$1.20 \cdot 10^{10}$	$2.17 \cdot 10^{10}$	$2.96 \cdot 10^{10}$	$6.53 \cdot 10^{10}$
BPE	12.12-12.68	11.72	57.28-3.86	19.70
PC	0.998	0.998	0.961	0.907
PQ	$7.44 \cdot 10^{-5}$	$4.11 \cdot 10^{-5}$	$4.38 \cdot 10^{-5}$	$1.87 \cdot 10^{-5}$

Table V

THE BLOCK COLLECTIONS AFTER BLOCK FILTERING.

terminates within a few minutes over all datasets.

Time Efficiency. We applied the basic and the advanced strategy of all techniques to the four datasets twice and measured the corresponding (average) $OTime$. The outcomes are presented in Table IV. Note that the basic approach was inapplicable to FR_C and FR_D, as its space requirements exceeded the available 4 TB of disk space.

For Stage 1, both Block Filtering strategies exhibit practically equivalent $OTime$, as the basic strategy offsets the cost of using two jobs by avoiding the computations repeated by the advanced one. However, the main reason for the equivalent overheads is the linear time complexity of Block Filtering and its simple functionality that processes large block collections at a negligible cost. Its exemplary performance also justifies the lack of a specialized load balancing for functions with linear complexity. The qualitative performance of Block Filtering is reported in Table V. Compared to Table II, the cardinality of all block collections is reduced by more than 60%, while their recall drops by less than 2%. As a result, the precision raises by 3 times, on average. The average number of blocks per entity is also significantly reduced, thus accelerating the computation of edge weights.

Stages 2 and 3 are treated as a whole in order to compare both strategies on an equal basis. When moving from left to right in Table IV, i.e., from the smallest block collection to the largest, the overhead time increases analogously for both strategies. Even for the largest dataset, though, the advanced strategy requires less than 12 hours in most cases, thus being dramatically faster than the serialized workflow, which requires almost 8 days over the high-end server described in the introduction.

Note also that there is a considerable variance between the efficiency of the two strategies, which designates that the parallelization of Meta-blocking is not a trivial task. The basic approach is consistently slower than the advanced one and their difference is particularly intense in the case of WEP and CEP. The inferior performance of the basic strategy should be attributed to the more jobs it employs and the higher I/O it yields between the independent nodes of the cluster: it creates a distinct edge for all comparisons, even the redundant ones, whereas the advanced approach creates a distinct edge only for the non-redundant comparisons; this means around 30% less comparisons in our datasets.

For the advanced strategy, CEP is the fastest algorithm in most cases, partly because it outputs the lowest number of comparisons. WEP follows in close distance, due to its similarly simple processing. CNP and WNP exhibit similar overhead times, as there are minor differences in the computational cost of their processing. They are the most time-consuming algorithms by far, since they process every edge twice, inside the neighborhoods of both adjacent nodes. They also retain significantly more comparisons than CEP and WEP, respectively (see RR below).

Qualitative Performance. To assess the quality of the restructured blocks produced by Meta-blocking, we consider the performance of the four pruning algorithms with respect to PC (recall), PQ (precision) and RR, namely the *Reduction Ratio*. RR expresses the relative decrease in the number of comparisons conveyed by Meta-blocking. Formally, $RR = 1 - ||B'||/||B||$, where B is the original and B' the restructured block collection [4]. For all measures, we estimated the average value and the standard deviation across the five weighting schemes per dataset. The outcomes are presented in Figures 8(a) to (c). In all diagrams, the higher a bar is, the better the corresponding performance.

Figure 8(a) demonstrates that the relative recall of the pruning algorithms remains the same across all datasets: the node-centric schemes, CNP and WNP, are more robust and

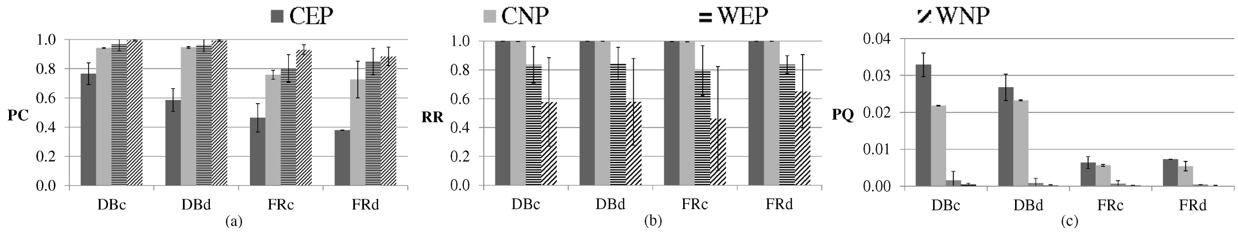

Figure 8. Average performance of the four pruning algorithms with respect to (a) *PC*, (b) *RR*, and (c) *PQ*.

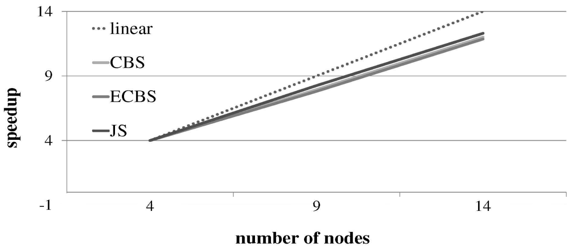

Figure 9. Scalability of advanced strategy for WEP over DB_C.

detect more duplicates than their edge-centric counterparts, CEP and WEP. The cardinality-based schemes, CEP and CNP, consistently achieve lower *PC* than the weight-based ones, WEP and WNP, which exceed 0.8 across all datasets. This means that they reduce the original *PC* by less than 10%, despite the significant enhancements in efficiency they convey. Indeed, Figure 8(b) shows that WEP achieves an *RR* close to 0.8, thus saving 80% of the original comparisons. The pruning of WNP is more shallow, as it retains at least one edge per node. Its *RR* fluctuates between 0.46 and 0.65, thus saving around half the original comparisons. For CEP and CNP, *RR* is consistently higher than 0.99. In fact, they perform such a deep pruning that they reduce the pairwise comparisons by 2 to 3 orders of magnitude across all datasets (still CNP retains twice as many comparisons as CEP, on average). This explains their poor recall. Yet, CEP and CNP achieve significantly higher precision across all datasets (Figure 8(c)). There is actually a trade-off between precision and recall for the four pruning algorithms: the higher *PQ* is for a specific method and dataset, the lower is the corresponding *PC* and vice versa. These patterns are in accordance with earlier findings about the relative performance of the pruning algorithms [17].

Scalability. Finally, we examined the scalability of the advanced strategy with respect to the available nodes. We applied all WEP to DB_C in combination with the 3 weighting schemes using 4, 9 and 14 slave nodes – including the master node. Figure 9 presents the speedup results along with the ideal case, in which the speedup is linear to the number of nodes (similar patterns were exhibited for the other datasets and are omitted for brevity). We observe that all the weighting schemes show a speedup close to the ideal one. For 14 slave nodes, it fluctuates between 11.8 and 12.3. This means that using more cluster nodes than we did in our experiments will improve the overhead time of the advanced strategy of Meta-blocking almost proportionally.

VI. Conclusions

We proposed two parallel versions of Meta-blocking based on MapReduce and equipped them with a load balancing algorithm that distributes the workload evenly among the cluster nodes. Our approaches dramatically increase the time efficiency of the serialized version, enabling blocking-based Entity Resolution in voluminous datasets. We observe that the basic parallelization strategy leads to significantly higher space requirements and is consistently slower than the advanced one, especially in the case of edge-centric algorithms. Our advanced strategy offers an optimized implementation that reduces the overhead of data exchange, leading to a speedup with the number of cluster nodes that is close to the ideal – regardless of the weighting scheme.

Acklowledgements. This work was partially supported by EU H2020 BigDataEurope (#644564) and PARTHENOS (#654119), and MEDA project (GSRTs KRIPIS action, Greece & European Regional Development Fund of EU).

References

[1] O. Benjelloun, H. Garcia, D. Menestrina, Q. Su, S. Whang, and J. Widom. Swoosh: a generic approach to entity resolution. *VLDB J.*, 18(1), 2009.
[2] I. Bhattacharya and L. Getoor. Collective entity resolution in relational data. *TKDD*, 1(1), 2007.
[3] C. Böhm, G. de Melo, F. Naumann, and G. Weikum. LINDA: distributed web-of-data-scale entity matching. In *CIKM*, 2012.
[4] P. Christen. A survey of indexing techniques for scalable record linkage and deduplication. *TKDE*, 24(9), 2012.
[5] P. Christen, T. Churches, and M. Hegland. Febrl - A parallel open source data linkage system. In *PAKDD*, 2004.
[6] V. Christophides, V. Efthymiou, and K. Stefanidis. *Entity Resolution in the Web of Data*. Synthesis Lectures on Data Management. Morgan & Claypool, 2015.
[7] A. Doan and A. Y. Halevy. Semantic integration research in the database community: A brief survey. *AI Magazine*, 26(1), 2005.
[8] X. Dong, A. Y. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. In *SIGMOD*, 2005.
[9] X. L. Dong and D. Srivastava. *Big Data Integration*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2015.
[10] V. Efthymiou, K. Stefanidis, and V. Christophides. Big data entity resolution: From highly to somehow similar entity descriptions in the Web. In *IEEE Big Data*, 2015.
[11] L. Getoor and A. Machanavajjhala. Entity resolution: Theory, practice and open challenges. *PVLDB*, 5(12), 2012.
[12] M. Herschel, F. Naumann, S. Szott, and M. Taubert. Scalable iterative graph duplicate detection. *IEEE Trans. Knowl. Data Eng.*, 24(11), 2012.
[13] H. Kim and D. Lee. Parallel linkage. In *CIKM*, 2007.
[14] L. Kolb, A. Thor, and E. Rahm. Dedoop: Efficient deduplication with hadoop. *PVLDB*, 5(12), 2012.
[15] L. Kolb, A. Thor, and E. Rahm. Multi-pass sorted neighborhood blocking with mapreduce. *Computer Science - R&D*, 27(1), 2012.
[16] G. Papadakis, E. Ioannou, T. Palpanas, C. Niederée, and W. Nejdl. A blocking framework for entity resolution in highly heterogeneous information spaces. *IEEE Trans. Knowl. Data Eng.*, 25(12), 2013.
[17] G. Papadakis, G. Koutrika, T. Palpanas, and W. Nejdl. Meta-blocking: Taking entity resolution to the next level. *IEEE Trans. Knowl. Data Eng.*, 2014.
[18] G. Papadakis, G. Papastefanatos, and T. Palpanas. Boosting the efficiency of large-scale entity resolution with enhanced meta-blocking. *Technical Report http://www.mi.parisdescartes.fr/%7Ethemisp/EMB-TR15.pdf*, 2015.
[19] S. E. Whang, D. Menestrina, G. Koutrika, M. Theobald, and H. Garcia-Molina. Entity resolution with iterative blocking. In *SIGMOD*, 2009.