http://dx.doi.org/10.1016/j.is.2016.12.001

Parallel Meta-blocking for Scaling Entity Resolution over Big Heterogeneous Data

Vasilis Efthymiou[1], George Papadakis[2], George Papastefanatos[3], Kostas Stefanidis[4], Themis Palpanas[5]

[1]*University of Crete, Greece & ICS-FORTH, Greece* *vefthym@ics.forth.gr*
[2]*University of Athens, Greece* *gpapadis@di.uoa.gr*
[3]*Athena Research Center, Greece* *gpapas@imis.athena-innovation.gr*
[4]*University of Tampere, Finland* *kostas.stefanidis@uta.fi*
[5]*Paris Descartes University, France* *themis@mi.parisdescartes.fr*

Abstract

Entity resolution constitutes a crucial task for many applications, but has an inherently quadratic complexity. In order to enable entity resolution to scale to large volumes of data, blocking is typically employed: it clusters similar entities into (overlapping) blocks so that it suffices to perform comparisons only within each block. To further increase efficiency, Meta-blocking is being used to clean the overlapping blocks from unnecessary comparisons, increasing precision by orders of magnitude at a small cost in recall. Despite its high time efficiency though, using Meta-blocking in practice to solve entity resolution problem on very large datasets is still challenging: applying it to 7.4 million entities takes (almost) 8 full days on a modern high-end server.

In this paper, we introduce scalable algorithms for Meta-blocking, exploiting the MapReduce framework. Specifically, we describe a strategy for parallel execution that explicitly targets the core concept of Meta-blocking, the blocking graph. Furthermore, we propose two more advanced strategies, aiming to reduce the overhead of data exchange. The comparison-based strategy creates the blocking graph implicitly, while the entity-based strategy is independent of the blocking graph, employing fewer MapReduce jobs with a more elaborate processing. We also introduce a load balancing algorithm that distributes the computationally intensive workload evenly among the available compute nodes. Our experimental analysis verifies the feasibility and superiority of our advanced strategies, and demonstrates their scalability to very large datasets.

Keywords: Meta-blocking, Map/Reduce Model, Parallelization

1. Introduction

Entity resolution (ER) is a very common task in Big Data processing, where different entity profiles, usually described under different schemas, are mapped to the same real-world object. Beyond the deduplication and cleaning problems that appear in traditional data integration, such as data warehouses, ER is a prerequisite for many Web applications, posing several challenges due to the volume and variety of the data collections. In general, ER constitutes an inherently quadratic task; given an entity collection, each entity profile must be compared to all others. Several approaches exist that aim to reduce the set of possible comparisons to be performed between two data collections [1]. *Blocking* is a typical method that reduces the number of pairwise comparisons by placing similar entity profiles into blocks and performing only the comparisons within each block.

Redundancy, i.e., placing every entity into multiple blocks, is employed by most blocking methods that handle noisy data [2, 3]. In fact, most blocking methods are *redundancy-positive* [4, 5]: the more blocks two entities share, the more likely they are to be matching. As an example, consider the simple approach of Token Blocking [6], which creates a block for every token that appears in the attribute values of at least two entities. Applying it to the entities in Figure 1(a), it yields the blocks in Figure 1(b), which place both pairs of matching entities, e_1-e_3 and e_2-e_4, in at least one block. Thus, they can be detected, despite their noisy, heterogeneous descriptions.

On the flip side, redundancy entails two kinds of unnecessary comparisons: the *rendundant* ones repeatedly compare the same entity profiles in multiple blocks, while the *superfluous* ones describe comparisons of profiles that are not matching. For example, the blocks b_2 and b_4 in Figure 1(b) contain one redundant comparison each, repeated in b_1 and b_3, respectively; given that entities e_1 and e_2 match with e_3 and e_4, respectively, the blocks b_5, b_6, b_7 and b_8 contain superfluous comparisons (the only exception is the redundant comparison e_3-e_5 in b_8, which is repeated in b_6). In total, the blocks of Figure 1(b) involve 13 comparisons, of which 3 are redundant and 8 superfluous. Such comparisons increase the computational cost without contributing any identified duplicates.

Current state-of-the-art. Numerous studies have focused on the problem of *block processing*, whose goal is to discard unnecessary (both redundant and superfluous) comparisons in order to enhance the precision of block collections. Most of the relevant techniques involve a functionality that operates at the block level, based on coarse-grained characteristics of the input block collection, such as the size of blocks: Block Purging [6] a-priori discards oversized blocks like b_8 in Figure 1(b), while Block Pruning [6] orders blocks from smallest to largest and terminates their processing as soon as the cost of identifying new duplicates exceeds a predefined threshold. Such techniques are

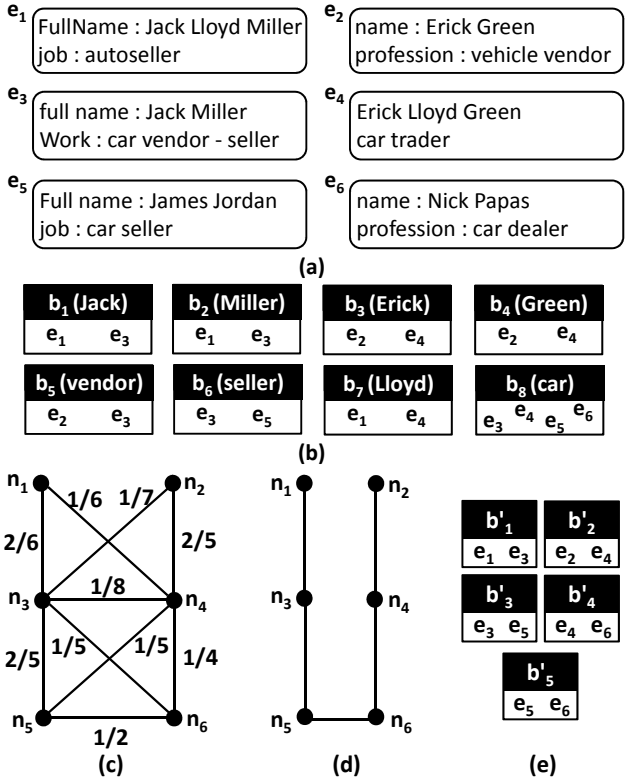


Figure 1: (a) A set of heterogeneous entity profiles [7], (b) the redundancy-positive block collection derived from them using Token Blocking, (c) the respective blocking graph that uses Jaccard similarity for edge weights, (d) one of the possible pruned blocking graphs, and (e) the restructured block collection after Meta-blocking.

efficient, but lack in accuracy, as their crude processing cannot control its impact on recall (in terms of matching comparisons).

A more fine-grained and accurate approach that results in significant time savings is *Meta-blocking* [4], which operates at the level of individual comparisons through a two-step procedure. First, it transforms the input block collection B into a *blocking graph* G_B. This graph contains a node n_i for every entity profile e_i in B and an edge $<n_i, n_j>$ for every pair of *co-occurring profiles* e_i and e_j, i.e., profiles that share at least one block. Every edge $<n_i, n_j>$ is associated with a weight $w_{i,j} \in [0, 1]$ that is analogous to the likelihood that the adjacent entities are matching – the higher an edge weight is, the more likely it corresponds to a matching comparison.[1] For instance, the blocking graph in Figure 1(c) is extracted from the blocks in Figure 1(b) using Jaccard similarity for weighting the edges. Note that two nodes cannot be linked by more than one edge, thus eliminating all redundant comparisons at no cost in recall (i.e., without missing any matching comparison).

In the second step, the edges with low weights are discarded according to a pruning criterion, so as to eliminate part of the superfluous comparisons. For instance, the pruned blocking graph in Figure 1(d) is derived from the graph in Figure 1(c) by discarding the edges with weight lower than the average one (1/4). The pruned blocking graph G'_B is then transformed into the restructured collection B' in Figure 1(e) by creating a new block for every retained edge. In total, the resulting blocks contain 5 comparisons, of which only 3 are unnecessary (superfluous). Compared to the initial blocks in Figure 1(b), the comparisons were reduced by 62% without any impact on recall. In our experiments, we show that Meta-blocking can reduce the time required to process the blocks and get the final matching results from 14 days to 95.5 minutes.

Scalability Limitations. The time complexity of Meta-blocking is linear with respect to the number of comparisons in the input block collection; it merely needs to iterate once or twice over the blocks in order to form and prune the blocking graph [4]. However, its running time depends on an additional parameter: the average number of blocks that are associated with every entity, called *Blocks Per Entity* (*BPE*). The reason is that the edge $<n_i, n_j>$ is weighted after estimating the intersection of the list of blocks associated with the profiles e_i and e_j. Therefore, the larger the input block collection is and the higher level of redundancy it involves, the more time-consuming the processing of Meta-blocking is.

So far, the largest dataset processed by Meta-blocking in the literature involves 3.4 million entities ($4.0 \cdot 10^{10}$ comparisons), each placed in 15 blocks on average (DB_C in Table 4). Using a high-end server with Intel i7 (3.40GHz), 64 GB of RAM and Debian Linux 7, the approach in Figures 1(c) and (d) requires 3 hours to execute. To assess the scalability of this approach, we tested it on a dataset with 7.4 million entities ($2.2 \cdot 10^{11}$ comparisons), each associated with 40 blocks on average (FR_D in Table 4). 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 (for comparison, we note that executing all pair-wise comparisons with a cheap string similarity metric, e.g., Jaccard similarity of the attribute value tokens, would require 50 days to complete).

Therefore, even as a pre-processing step for ER, Meta-blocking is a heavy computational task with serious efficiency limitations at the scale of Web data. We expect this problem to aggravate in the future, as Web data continuously grow, both in terms of number of entity profiles and in terms of amount of information inside each profile[2]. To overcome these limitations of the existing serialized techniques, novel distributed approaches are required.

Proposed Solution. In this paper, we adopt the MapReduce programming model [8] for parallelizing Meta-blocking and scaling its techniques to voluminous Web data collections, with no impact on its effectiveness. We provide three parallelization strategies.

The first one explicitly targets the blocking graph, which builds and stores all the edges along with their weights, and prunes the ones with the lowest weights. Even though MapRe-

[1] Theoretically, the computational cost of estimating the edge weights of the blocking graph G_B is equivalent to executing all pairwise comparisons in the input blocks B. In practice, though, the weight computation for an edge $<n_i, n_j>$ requires a fraction of the time taken by a string similarity metric to compare the corresponding entities n_i-n_j: the former merely compares two lists of integers, the block ids associated with n_i and n_j, while the latter compares the textual attribute values of n_i and n_j [4, 5].

[2] `http://stats.lod2.eu`

duce leads to a significant speedup through the parallel processing of the blocking graph, it bears a significant I/O cost that may become a bottleneck, when building very large blocking graphs.

The second strategy overcomes this shortcoming, offering a more efficient implementation, which uses the blocking graph implicitly. In particular, it involves a pre-processing task, which enriches the input block collection with indexing information used for computing the weights of the edges, without building and storing any of them explicitly. Then, a final processing task is responsible for pruning the low-weighted comparisons (according to the chosen pruning scheme).

The third strategy offers an alternative approach that is independent of the blocking graph. At its core lies the idea that the main information required for the estimation of edge weights is the number of blocks shared by the adjacent entities. Thus, for every entity, it aggregates the bag of all entities that co-occur with it in at least one block. Then, it derives the edge weight that corresponds to a specific neighbor from its frequency in the co-occurrence bag. To minimize its I/O cost, this approach does not store its edge weights on the disk. In cases where they do not need to be recomputed, this strategy results in very few Map/Reduce jobs that are efficiently executed.

In order to avoid potential bottlenecks associated with the computation-intensive parts of our MapReduce functions, we also introduce a novel load balancing algorithm, called *MaxBlock*. It exploits the highly-skewed distribution of block sizes in redundancy-positive collections in order to split them in partitions of equivalent computational cost (i.e., total number of comparisons). We experimentally compare MaxBlock with existing approached, including a state-of-the-art algorithm that serves a similar purpose, and demonstrate that our approach has significant qualitative and quantitative benefits.

Finally, we provide an extensive experimental evaluation of our methods over the Hadoop[3] environment. We apply all Meta-blocking configurations to four large-scale, real-world datasets measuring the corresponding running times. The outcomes verify the high efficiency of our distributed techniques. To facilitate other researchers to experiment with parallel Meta-blocking, we have publicly released both the data and the implementation of our methods[4].

Contributions. In summary, this paper makes the following contributions:

- We adapt Meta-blocking to the MapReduce paradigm through 3 alternative parallelization strategies: an edge-based strategy that explicitly builds the blocking graph, a comparison-based strategy that uses the blocking graph implicitly, as a conceptual model, and an entity-based strategy that is independent of the blocking graph. We also provide concrete implementations for all weighting schemes that are used in Meta-blocking.
- We present *MaxBlock*, a load balancing algorithm that deals with skewness in the input block collection, splitting it into partitions of the same computational cost. Our experiments verify that it has qualitative and quantitative advantages over a state-of-the-art solution.
- We verify the scalability of our techniques through a thorough experimental evaluation over the four largest, real datasets that have been applied to Meta-blocking. The data and the implementation of our techniques are publicly available.

The rest of the paper is organized as follows[5]: Section 2 provides the preliminaries for blocking and Meta-blocking. We outline our adaptation of Meta-blocking to the MapReduce paradigm in Section 3 and elaborate on the parallelization of each stage in Sections 4 to 9. Section 10 presents our algorithm for load balancing and Section 11 evaluates the performance of our approaches. Finally, Section 12 describes the related work, and Section 13 concludes the paper with a summary of our findings along with plans for future work.

2. Preliminaries

Blocking. The core notion of Entity Resolution is the *entity profile*, or simply entity, which comprises a set of name-value pairs that are uniquely identified through a global id. We denote an entity with id i by e_i. Two entities that correspond to the same real-world object are called *duplicates* or *matches*. A set of entities is called *entity collection* (E) and $D(E)$ stands for the set of duplicate entities it contains. $|D(E)|$ denotes the corresponding set size, i.e., the number of duplicate pairs in E.

Blocking aims to tame the quadratic complexity of Entity Resolution by restricting the executed comparisons to entities that are similar in some respect. Given an entity collection, a blocking method groups its entities into clusters that are called *blocks*. We denote an individual block with id i by b_i. The number of entities it contains is called *block size* ($|b_i|$), while the number of comparisons in it is called *block cardinality* ($\|b_i\|$). Apparently, $\|b_i\| = |b_i| \times (|b_i| - 1)/2$.

Collectively, a set of blocks is called *block collection* (B). Its size is denoted by $|B|$ and represents the total number of blocks it contains. $\|B\|$ stands for its *total cardinality*, which is equal to 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 and e_j that are placed within the same block are called *co-occurring*. Their comparison is denoted by $c_{i,j}$, while $B_{i,j}$ represents the set of blocks they co-occur in. Its size, $|B_{i,j}|$, stands for the number of blocks shared by e_i and e_j. If e_i and e_j are duplicates, $c_{i,j}$ is a *matching comparison*.

Regarding the performance of a block collection, a common premise in the literature is that it is independent of the entity matching method that executes the pair-wise comparisons [4, 2, 10]. The rationale is that two duplicates can be 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. The goal of blocking is to maximize the number of co-occurring duplicates ($|D(B)| \subseteq |D(E)|$), while minimizing the number of executed comparisons ($\|B\|$).

More formally, two established measures are used for assessing the performance of a block collection [11, 2, 3, 10]:

[3] `https://hadoop.apache.org`

[4] See `https://github.com/vefthym/ParallelMetablocking`

[5] A preliminary abridged version of this paper appeared in [9].

Aggregate Reciprocal Comparisons Scheme (**ARCS**)	$ARCS(e_i, e_j, B) = \sum_{b_k \in B_{ij}} \frac{1}{\|b_k\|}$
Common Blocks Scheme (**CBS**)	$CBS(e_i, e_j, B) = \|B_{ij}\|$
Enhanced Common Blocks Scheme (**ECBS**)	$ECBS(e_i, e_j, B) = CBS(e_i, e_j, B) \cdot \log\frac{\|B\|}{\|B_i\|} \cdot \log\frac{\|B\|}{\|B_j\|}$
Jaccard Scheme (**JS**)	$JS(e_i, e_j, B) = \frac{\|B_{ij}\|}{\|B_i\| + \|B_j\| - \|B_{ij}\|}$
Enhanced Jaccard Scheme (**EJS**)	$EJS(e_i, e_j, B) = JS(e_i, e_j, B) \cdot \log\frac{\|V_B\|}{\|n_i\|} \cdot \log\frac{\|V_B\|}{\|n_j\|}$

Figure 2: The formal definition of the weighting schemes of Meta-blocking. For EJS, $|V_B|$ stands for the order (i.e., number of nodes) of the blocking graph G_B, while $|n_x|$ denotes the degree of node n_x.

• *Pairs Completeness* (*PC*) is analogous to recall, estimating the portion of existing pairs of duplicates that are co-occurring: $PC = |D(B)|/|D(E)|$. *PC* 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: $PQ = |D(B)|/\|B\|$. *PQ* is defined in the interval $[0, 1]$, with higher values indicating better precision. Note that *PQ* provides a pessimistic estimation of precision, because it considers as true positives only the non-redundant matches; the redundant comparisons that involve duplicates are regarded as false positives.

Ideally, the goal of blocking is to maximize both *PC* and *PQ*. However, there is a clear trade-off between the two measures: the more comparisons are contained in B (higher $\|B\|$), the more duplicates are co-occurring (higher $|D(B)|$) and the higher the recall (*PC*) gets. Given, though, that $\|B\|$ increases quadratically for a linear increase in $|D(B)|$ [12, 13], precision (*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. They try to maximize precision, while maintaining recall at very high levels, above 0.80 or even 0.90, such that blocking places the vast majority of the matches in at least one common block.

Meta-blocking. Most of the blocking methods are *redundancy-positive* in the sense that their blocks provide positive evidence for the matching likelihood of two entities: the more blocks they share, the more likely they are to match [4, 5]. The main characteristic of redundancy-positive blocking methods is that they trade very high *PC* for very low *PQ*. In other words, they yield a large number of unnecessary comparisons in their effort to achieve high recall.

Meta-blocking operates on their block collections in order 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 $PQ(B') \gg PQ(B)$ [4, 5]. It targets individual comparisons and a-priori discards the unnecessary ones (superfluous or redundant) by pruning edges of the blocking graph. Its performance depends on two parameters: the weighting and the pruning scheme.

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. The following approaches have been proposed in the literature (their weights are restricted to $[0, 1]$ through normalization) [4]:

• *Aggregate Reciprocal Comparisons Scheme* (ARCS) is based on the assumption that the smaller the blocks two entities share, the more likely they are to be matching.

• *Common Blocks Scheme* (CBS) captures the fundamental property of redundancy-positive block collections that the more blocks two entities share, the more likely they are matching.

• *Enhanced Common Blocks Scheme* (ECBS) improves CBS by discounting the contribution of the entities that participate in many blocks.

• *Jaccard Scheme* (JS) estimates the portion of blocks shared by two entities.

• *Enhanced Jaccard Scheme* (EJS) improves JS by discounting the contribution of entities involved in too many non-redundant comparisons (i.e., high node degree).

The formal definitions of the weighting schemes are presented in Figure 2.

The second parameter, the *pruning scheme*, relies on a *pruning criterion* which can be either weight- or cardinality-based; the former specifies the minimum weight of the retained edges and the latter the maximum number of retained edges. With respect to its scope, the pruning criterion can be either global, applying to the entire blocking graph, or local, covering an individual node neighborhood. The selected criterion is then combined with a *pruning algorithm*, which is either edge- or node-centric; the former iterates over all edges of the graph to retain the globally best ones and the latter over all edges of the neighborhood to retain the locally best ones. On the whole, the main pruning schemes are the following [4]:

• *Weighted Edge Pruning* (WEP) combines the edge-centric algorithm with a global weight threshold that amounts to the average edge weight of the entire blocking graph. Thus, it retains all edges with a weight higher than the overall mean one.

• *Cardinality Edge Pruning* (CEP) couples the edge-centric algorithm with a global cardinality threshold. Thus, it 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) combines the node-centric pruning algorithm with a local weight threshold that amounts to the average edge weight of each neighborhood.

• *Cardinality Node Pruning* (CNP) combines the node-centric pruning algorithm with a global cardinality threshold. For each neighborhood, it retains the top-k edges, with $k = \lfloor \sum_{b_i \in B} |b_i|/|E| - 1 \rfloor$.

In this work, we consider all weighting and pruning schemes, adapting their functionality to the MapReduce paradigm. All of them benefit greatly from the pre-processing technique of *Block Filtering* [7], which drastically reduces the size of the blocking graph. Similar to Meta-blocking, it transforms a redundancy-positive block collection B into a new one B' that involves a lower number of 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 (i.e., higher $\|b_i\|$), the less important it is for its entities.

In more details, 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). For

Name	Symbol
Entity collection	E
Duplicate entities in E	$D(E)$
Entity profile with id i	e_i
Block collection	B
Block collection size (number of blocks)	$\|B\|$
Block collection cardinality (number of comparisons)	$\|\|B\|\|$
Blocks containing e_i	B_i
Number of blocks containing e_i	$\|B_i\|$
Block with id i	b_i
Block size (number of entities)	$\|b_i\|$
Block cardinality (number of comparisons)	$\|\|b_i\|\|$
Blocks shared by e_i and e_j	$B_{i,j}$
Number of blocks shared by e_i and e_j	$\|B_{i,j}\|$
Comparison between e_i and e_j	$c_{i,j}$
Duplicate entities in B	$D(B)$
Blocking graph	G_B
Node in G_B corresponding to e_i	n_i
Edge in G_B	$< n_i, n_j >$
Weight of $< n_i, n_j >$	$w_{i,j}$

Table 1: Summary of the notation used in this work.

every entity e_i, this threshold is locally defined as $N_i = \lfloor r \times |B_i| \rfloor$, where $r \in [0, 1]$ 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 increase efficiency to a significant extent, pruning at least 50% of the blocking graph's edges, while having a negligible impact on recall [7].

Table 1 summarizes the notation used in the rest of the paper.

3. Approach Overview

In the following, we elaborate on the adaptation of Meta-blocking to MapReduce. The MapReduce programming model consists of two consecutive procedures that can be grouped into *jobs*: first, the *Map* phase receives a set of (`key`, `value`) pairs and transforms it into a new output set of pairs. Second, the *Reduce* phase receives a set of (`key`, `value`) pairs that share the same `key` and are sorted according to their `value`; it performs a summary operation on them to produce a new, usually smaller set of pairs.

The serialized workflow we want to parallelize is depicted in Figure 3(a) and 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 is presented in Figure 3(b) and consists of three stages. Again, the first one applies Block Filtering to the input block collection and the last one implements Meta-blocking. The only difference is in the second stage, which preprocesses the blocks in order to transform them into a suitable form for parallel Meta-blocking.

We analyse every stage of the parallelized workflow separately, proposing at least two different approaches in each case. The first one applies a basic strategy that relies on a straightforward adaptation, but involves more jobs and higher I/O between the nodes. The other approach(es) correspond(s) to more advanced strategy(ies), reducing the overhead of data exchange through more elaborate processing. In all cases, we provide the pseudo-code of the strategy's functionality and, for the most important strategies, we accompany it with an example that facilitates its understanding.

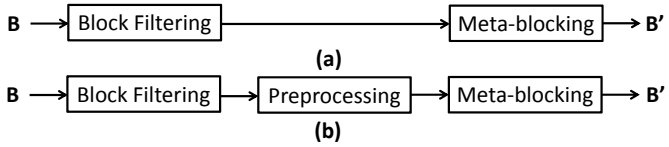


Figure 3: (a) The serialized workflow of Meta-blocking, and (b) its parallelized counterpart.

Section 4 presents two strategies for the first stage (Block Filtering), while Section 5 introduces three strategies for the second stage (Preprocessing). The last stage of the parallel workflow applies one of the four pruning algorithms 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 examine every pruning algorithm separately, in Sections 6 to 9. Given that the functionality and the complexity of their parallelization depend on the preprocessing strategy, we present three adaptations in every section, each of them corresponding to the output of the previous stage. Finally, Section 10 introduces a novel algorithm for load balancing that applies to all strategies of the last two stages.

Note that in every stage, special care was taken to minimize the I/O between the independent nodes. Part of this effort focused on *optimizing our representation model*. Apparently, we could use the actual blocking keys and URIs to identify the blocks and the entities, respectively. However, the binary representation of these textual values is much larger than that of numerical identifiers. For this reason, our model relies exclusively on numbers: we enumerate every block and entity, so that they are uniquely identified by an integer id, and represent the edges by the concatenation of the adjacent entity ids. Their weights are naturally represented by real numbers.

4. Stage 1: Block Filtering

The first stage applies Block Filtering to the input block collection 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 stands for the id of the block, while the value contains the list of the entity ids placed in b_k:

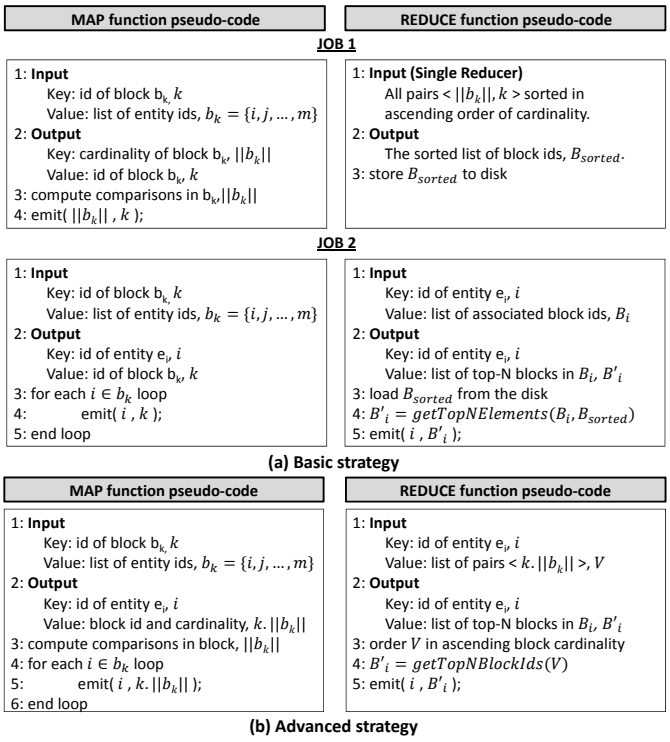


Figure 4: Pseudo-code interpretation of (a) the basic and (b) the advanced strategy for Block Filtering. They employ a global and a local ordering of blocks, respectively.

key=k and value=$\{i, j, \ldots, m\}$ for b_k=$\{e_i, e_j, \ldots, e_m\}$. 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 ids of the blocks still containing e_i: key=i and value=B'_i.

4.1. Basic Strategy

This strategy employs two MapReduce jobs. The first one sorts all blocks globally in ascending order of cardinality, producing the sorted list B_{sorted}. The second job uses B_{sorted} in order to identify the most important blocks for each entity.

The functionality of the first job is outlined in the upper part of Figure 4(a). 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 pseudo-code interpretation of the second job is presented in the lower part of Figure 4(a). The `map` function gets the same input as the first job: the id of a block along with the entity ids it contains. For every entity e_i contained in the given block b_k, it emits as output a pair (i, k). MapReduce groups together all pairs having the same key 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 the emitted as output: key=i, value=B'_i.

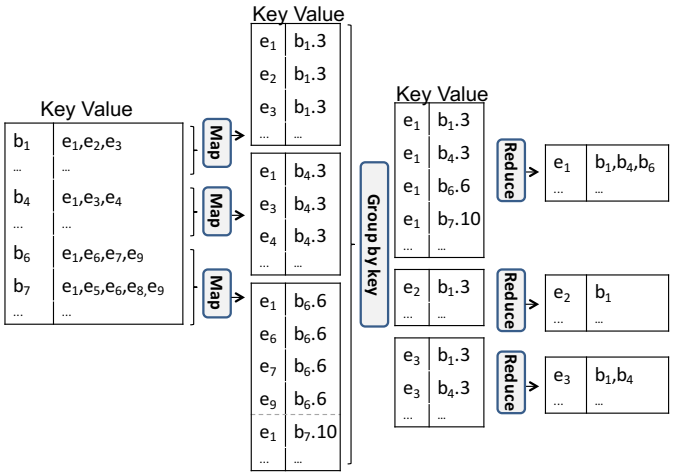


Figure 5: An example of the advanced strategy for Block Filtering.

4.2. Advanced Strategy

The rationale behind the advanced strategy is to use a single MapReduce job that provides the `reduce` function with the necessary information for sorting the blocks of each entity locally. Its functionality is outlined in Figure 4(b). The `map` function gets as input the id and the entities of a block b_k and computes 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 5 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.

5. Stage 2: Preprocessing

The second stage of the parallel Meta-blocking workflow prepares the input that will be processed by the selected pruning algorithm in the third stage. It plays a crucial role, as its output 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 three different strategies for Preprocessing, which share the same input (i.e., the outcome of Block Filtering), but differ in their output. The *edge-based* strategy explicitly creates the blocking graph, performing all

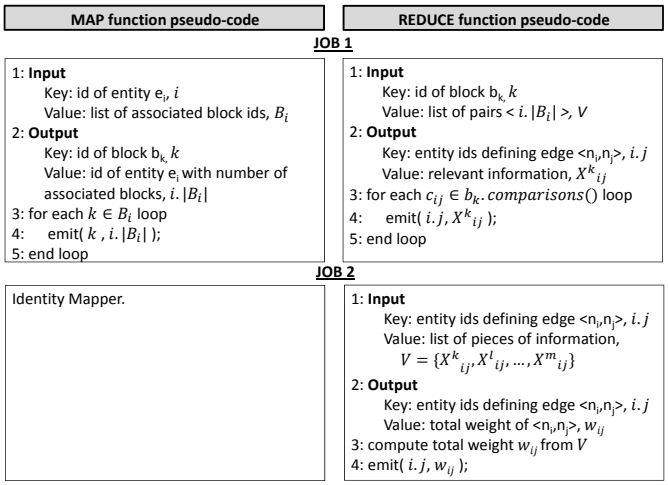


Figure 6: Pseudo-code interpretation of the edge-based Preprocessing strategy, which explicitly creates the blocking graph.

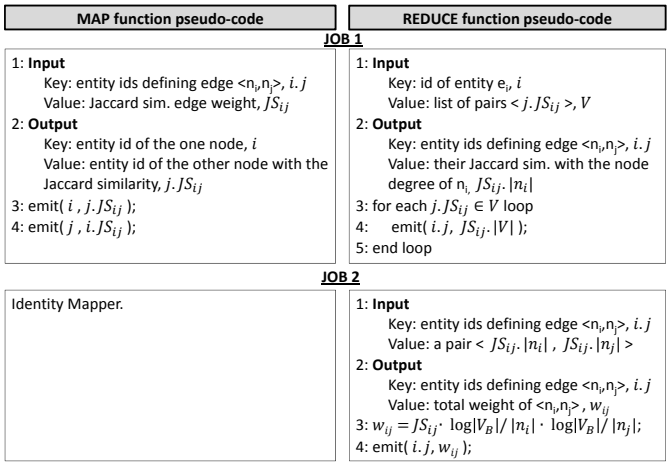


Figure 7: Pseudo-code interpretation of the edge-based Preprocessing strategy for the EJS weighting scheme.

weight computations in order to simplify the functionality of the pruning algorithm. On the flip side, it involves two MapReduce jobs with high I/O that store all edges to the disk. The *comparison-based* strategy defers all weight computations and simply facilitates them by enriching the input of the pruning algorithms with all the necessary information. The *entity-based* strategy facilitates a different approach for weight estimation that does not require any preprocessing. Thus, it simply receives the output of Block Filtering (i.e., the block ids retained per entity) and transforms it into a new block collection. Due to their simplicity, the last two strategies require just one job.

5.1. Edge-based Strategy: Explicit Blocking Graph

The pseudo-code interpretation of the edge-based strategy is depicted in Figure 6. The first MapReduce 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 $|B_i|$ denotes the number of blocks containing e_i after Block Filtering. The reason is that $|B_i|$ is the cornerstone for most weighting schemes.

The `reduce` function of the first job groups together all entities contained in a block b_k and is able to reproduce all its comparisons.[6] For every comparison between entities e_i and e_j (c_{ij}), it emits the concatenation of their ids as key and some local information X^k_{ij} as value: `key`=$i.j$ and `value`=X^k_{ij}. The information in X^k_{ij} is necessary for estimating the corresponding edge weight and varies, depending on the selected weighting scheme. For ARCS, it comprises the cardinality of block b_k (i.e., X^k_{ij}=$\|b_k\|$), while for all other schemes it concatenates $|B_i|$ and $|B_j|$ (i.e., X^k_{ij}=$|B_i|.|B_j|$); for CBS, though, it can be empty.

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 all local information pertaining to the edge $<n_i, n_j>$ that is specified by the input key. Based on them, the reducer computes the corresponding edge weight w_{ij} from the formulas in Figure 2. For example, we simply have $w_{ij} = |V|$ for CBS, as the size of the value list equals the number of common blocks, $|B_{ij}|$. As output, the reducer emits a pair with the id and the weight of the edge: `key`=$i.j$ and `value`=w_{ij}.

There is an exception to this strategy, as the EJS weighting scheme requires two additional jobs to be applied to the output of JS. Their goal is to estimate the node degree $|n_i|$ of every entity e_i. The first job counts the edges that are adjacent to each entity, while the second one reassembles all neighboring entities in order to estimate the weight of their adjacent edge according to EJS formula in Figure 2. Their functionality is outlined in Figure 7.

The first job continues from the Preprocessing of the JS weighting scheme. Its `map` function receives as input an individual edge from the respective blocking graph; the concatenated ids of the adjacent entities form the key, while the value contains the corresponding edge weight. The mapper performs no processing, but just emits two pairs: for each of the two entities, it uses its id as the key and concatenates the id of the other entity with the edge weight to form the value.

In this way, the `reduce` function gathers all edges that correspond to a specific entity e_i. Its input value actually comprises a list with the ids of all neighboring entities appended to the weight of the respective edge. The size of this list equals the degree $|n_i|$ of node n_i that corresponds to e_i. The reducer emits this information so that the corresponding EJS weights can be computed in the second job: for each of the neighboring entities e_j, it emits a pair with `key`=$i.j$ and `value`=$JS_{ij}.|n_i|$.

The second job involves an identity mapper so that the reducer gathers both values that pertain to an individual edge $<n_i, n_j>$, namely $JS_{ij}.|n_i|$ and $JS_{ij}.|n_j|$. Having this information, the EJS weight can be derived from the Formula in Figure 2. This forms the output value, while the ids of the adjacent entities form the output key.

[6] Note that a block with just one remaining entity contains no comparison and, thus, no processing is performed.

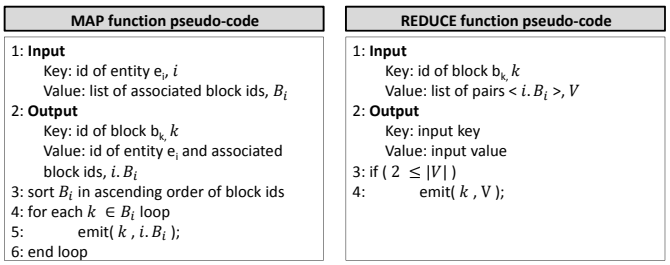


Figure 8: Pseudo-code interpretation of the comparison-based Preprocessing strategy, which creates the blocking graph implicitly, enriching the description of the input blocks with the necessary information for weight estimation.

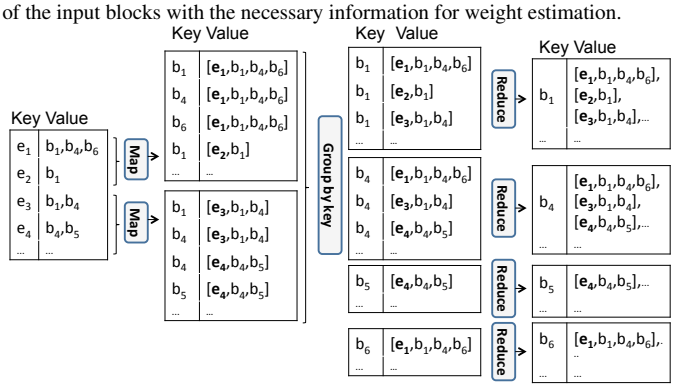


Figure 9: An example of the comparison-based strategy for Preprocessing.

5.2. Comparison-based Strategy: Implicit Blocking Graph

This strategy creates the blocking graph implicitly: it enriches the description of the input block collection with the information that is required for detecting all edges and estimating their weights according to the selected scheme. The key to this approach is the idea that every edge $<n_i, n_j>$ of the blocking graph G_B corresponds to a non-redundant comparison c_{ij} in the block collection B.

A comparison c_{ij} in b_k is *non-redundant* only if it satisfies the Least Common Block Index condition (LeCoBI for short). That is, if the id of b_k equals the least common block id of the entities e_i and e_j: $k = min(B_i \cap B_j)$ [14]. 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. The comparison-based strategy integrates this information to its output, so that the pruning algorithms carry out all edge and weight computations in the third stage.

This functionality is performed by one MapReduce job, which is outlined in Figure 8. The `map` function receives as input the outcome of Block Filtering: 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; provided that there are at least two entities, it emits the same (`key, value`) pair as output: `key`=k and `value`=$\{i.B_i, j.B_j, \ldots, m.B_m\}$.

Figure 9 provides an example of this functionality. For each

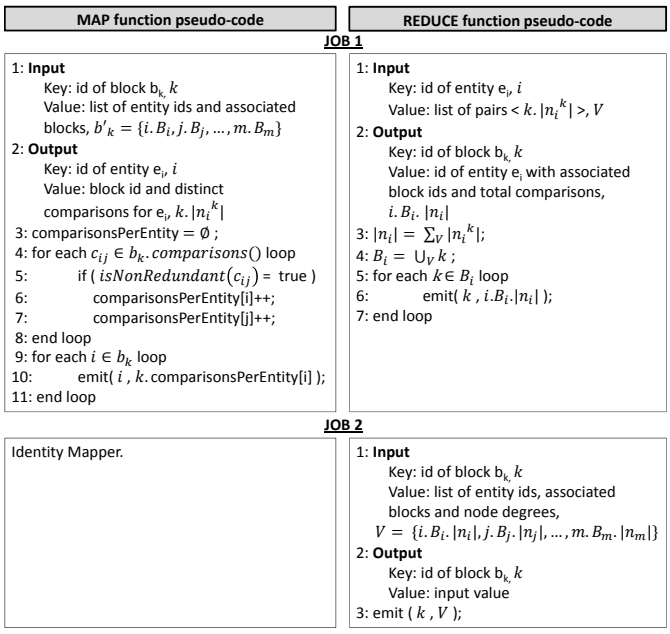


Figure 10: Pseudo-code interpretation of the comparison-based strategy for Preprocessing the EJS weighting scheme.

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.

There are two exceptions to this functionality, because the weighting schemes ARCS and EJS need additional information for estimating their edge weights. The former requires the cardinality of every block contained in B_i. This can be easily embedded into the output of the previous stage (Block Filtering), such that for ARCS, the input is not only a list of associated blocks for each entity, but also the cardinality of each such block. Then, the rest of the process is the same as in the other weighting schemes. In contrast, the information required by EJS can only be derived from an elaborate processing.

In more detail, the EJS weighting scheme requires the degree of the nodes corresponding to the adjacent entities of every edge. The main idea is that the degree of node n_i equals the total number of non-redundant comparisons involving the corresponding entity e_i. This can be assessed through a single job that applies to the output of the job in Figure 8: it estimates the partial non-redundant comparisons per entity in the mapper and the total ones in the reducer. A second job is then used to reconstruct the enriched description of the blocks. Their functionality is depicted in Figure 10.

The first `map` function receives as input the enriched description of a block b_k: the block id k is the key, while the value contains the corresponding entities together with their associated blocks. For each of these entities, it creates a comparison counter (Line 3). It iterates over all comparisons in b_k and for those that satisfy the LeCoBI condition, it increases the counters of the involved entities (Lines 4-8). Finally, for every entity,

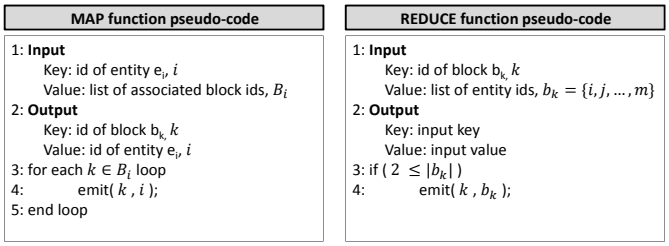


Figure 11: Pseudo-code interpretation of the entity-based strategy for Preprocessing, which does not use the blocking graph.

it emits a pair with its id as key and the block id concatenated with the comparison counter as value (Lines 9-11).

In this way, the first `reduce` function gathers all pairs pertaining to a specific entity e_i. Its id is the input key, while the input value comprises a list of all pairs $k.|n_i^k|$, where k is the id of a block containing e_i and $|n_i^k|$ is its partial node degree in b_k (i.e., the number of non-redundant comparisons of b_k that involve e_i). Thus, the reducer is able to estimate the total node degree of e_i (Line 3). It also reconstructs the list of the blocks associated with e_i, B_i (Line 4). For each block $b_k \in B_i$, it then emits a pair with `key`=k and `value`=$i.B_i.|n_i|$ (Lines 5-7).

The second job aims to reconstruct the enriched input of the blocks. Thus, it contains an identity mapper that forwards its input to the reducer, which aggregates all information pertaining to an individual block b_k; the block id is the key, while the value contains the ids of all entities in b_k, their associated blocks as well as the corresponding node degrees. This is also the output of the reducer, which is emitted without any further processing.

5.3. Entity-based Strategy: No Blocking Graph

This strategy is fundamentally different from the others in the sense that it does not require the blocking graph. Its functionality revolves around the individual entities such that for every entity e_i, it estimates the weights of its co-occurring entities with a single iteration over the contents of the associated blocks B_i. To facilitate this procedure, the Pre-processing stage simply transforms the output of Block Filtering into a block collection.

This is performed with a single MapReduce job that is presented in Figure 11. The `map` function receives the id of an entity e_i as input key and the associated blocks B_i as input value. For every block $b_k \in B_i$, it simply emits its id k as the key and the id of the entity i as value. Then, MapReduce groups together all pairs with the same key, thus reassembling all blocks. In more detail, the `reduce` function receives as input key the id k of a specific block $b_k = \{e_i, e_j, \ldots, e_m\}$, while the input value comprises the ids of the corresponding entities: `value`=$\{i, j, \ldots, m\}$. Without any further processing, it emits the same `(key, value)` pair as output.

6. Stage 3: Weighted Node Pruning (WNP)

WNP refines the blocking graph by processing every node neighborhood independently of the others. For every node, it discards the incident edges that have a weight lower than the average edge weight of the neighborhood. We propose three parallelization strategies for this algorithm – one for every Preprocessing strategy. They all require a single MapReduce job.

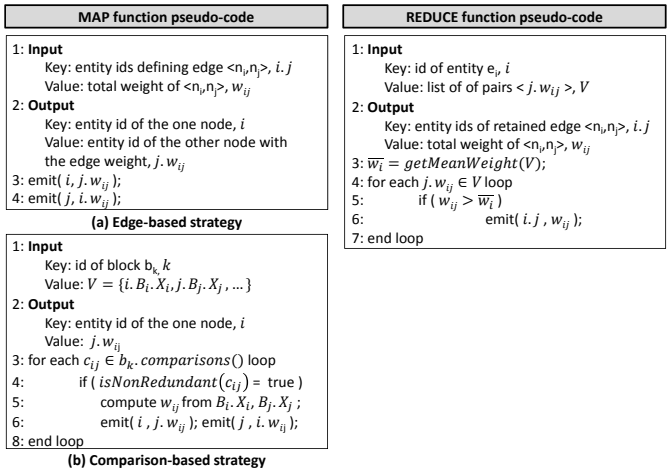


Figure 12: Pseudo-code interpretation of (a) the edge-based and (b) the comparison-based strategy for WNP. They share the same `reduce` function.

6.1. Edge-based Strategy

The functionality of this strategy is outlined in Figure 12 – together with the comparison-based strategy. They share the same reducer, but they differ in the mapper, due to the different input they receive.

In more detail, Figure 12(a) depicts the edge-based `map` function. It takes as input key the id of an individual edge $<n_i, n_j>$ ($i.j$) and as input value the corresponding weight (w_{ij}). 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 ($j.w_{ij}$ or $i.w_{ij}$).

The `reduce` function in Figure 12 receives as input key the id i of an entity e_i that defines a neighborhood 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, $\bar{w}_i$, in Line 3. Then, in Lines 4-7, it iterates over all adjacent edges and for every edge $<n_i, n_j>$ with a weight higher than the average one, it emits a pair ($i.j$, w_{ij}).

6.2. Comparison-based Strategy

The `map` function of this strategy appears in Figure 12(b). It operates on the enriched description of an individual block b_k: the input key contains its id (k), while the input value is of the form `value` = $\{i.B_i.X_i, j.B_j.X_j, \ldots\}$; that is, it comprises the corresponding entity ids, the blocks ids associated with each entity and the local information required by the selected weighting scheme. For EJS, this information contains the node degree (X_i=$|n_i|$), for ARCS it contains the cardinalities of the blocks in B_i (X_i=$\{||b_j|| : b_j \in B_i\}$) and for all other weighting schemes it is empty. The mapper iterates over all comparisons in the given block (Line 3). For every non-redundant comparison, it computes the corresponding edge weight from the associated block ids and the local information X_i of the weighting scheme (Lines 4-5). Then, it emits two `(key, value)` pairs, one for

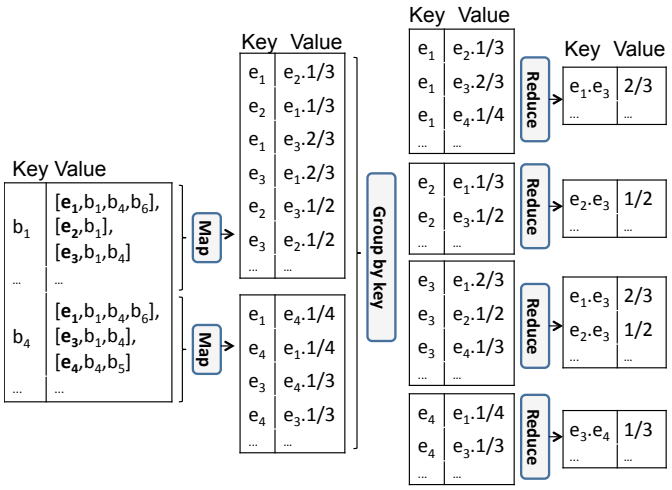
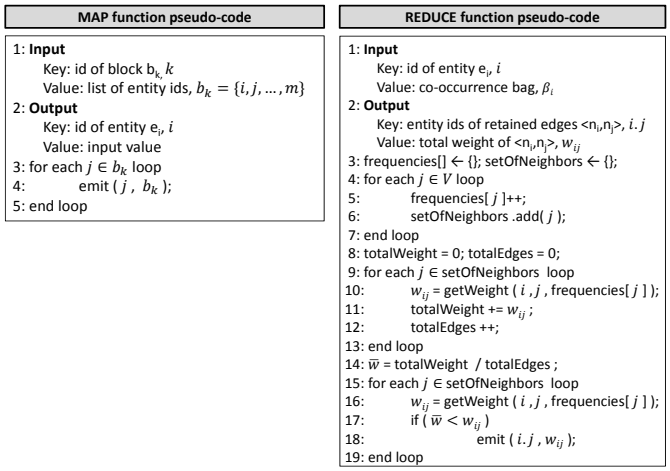


Figure 13: An example of the comparison-based strategy for WNP, using the JS weighting scheme.



MAP function pseudo-code

```
1: Input
     Key: id of block b_k, k
     Value: list of entity ids, b_k = {i, j, ..., m}
2: Output
     Key: id of entity e_i, i
     Value: input value
3: for each j ∈ b_k loop
4:         emit ( j , b_k );
5: end loop
```

REDUCE function pseudo-code

```
1: Input
     Key: id of entity e_i, i
     Value: co-occurrence bag, β_i
2: Output
     Key: entity ids of retained edges <n_i,n_j>, i.j
     Value: total weight of <n_i,n_j>, w_ij
3: frequencies[] ← {}; setOfNeighbors ← {};
4: for each j ∈ V loop
5:         frequencies[ j ]++;
6:         setOfNeighbors .add( j );
7: end loop
8: totalWeight = 0; totalEdges = 0;
9: for each j ∈ setOfNeighbors loop
10:        w_ij = getWeight ( i , j , frequencies[ j ] );
11:        totalWeight += w_ij ;
12:        totalEdges ++;
13: end loop
14: w̄ = totalWeight / totalEdges ;
15: for each j ∈ setOfNeighbors loop
16:        w_ij = getWeight ( i , j , frequencies[ j ] );
17:        if ( w̄ < w_ij )
18:                emit ( i.j , w_ij );
19: end loop
```

Figure 14: Pseudo-code interpretation of the entity-based strategy for WNP.

each of the adjacent entities (Line 6) – just like the mapper in Figure 12(a).

Figure 13 shows an example that applies this functionality in combination with the JS weighting scheme to the output of Figure 9. 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` function, 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.

6.3. Entity-based Strategy

The outline of this strategy appears in Figure 14. Its `map` function receives as input key the id k of block b_k and as input value the entity ids contained in b_k. For every entity $e_i \in b_k$, it simply emits its id as key (i.e., `key`=i) and the entire block b_k as value (i.e., `value`=b_k). In this way, the reducer aggregates the *co-occurrence bag* of entity e_i, i.e., the ids of all entities that share at least one block with e_i. The frequency of an entity e_j in this bag amounts to $|B_{ij}|$, the number of blocks it shares with e_i. This is the core information required by all weighting schemes for estimating the corresponding edge weight w_{ij}.

Based on this rationale, the `reduce` function estimates the edge weights using two data structures, which are initialized in Line 3: the array $frequencies$, which gathers the number of appearances of each entity, and the set $setOfNeighbors$, which aggregates the ids of the distinct co-occurring entities. For every entity in the co-occurrence bag, the reducer updates its frequency in the array and adds it to the set of neighbors (Lines 4-7). Subsequently, it estimates the weights of the edges incident to e_i from the distinct neighbors in $setOfNeighbors$ (Lines 9-10). At the same time, it derives the average weight of the neighborhood, $\bar{w}$, with the help of two counters (Lines 8 & 11-14). The final loop in Lines 15-19 repeats the estimation of edge weights and retains those exceeding $\bar{w}$; the ids of their adjacent entities are emitted as keys and their weights as values.

Note that after the loop in Lines 4-7, $setOfNeighbors$ contains the id of the neighborhood's center, e_i. Given that e_i co-occurs with itself in all blocks, its edge weight would be equal (or close) to 1 for all weighting schemes. To avoid retaining the meaningless comparison $c_{i,i}$ and to avoid distorting the weight threshold $\bar{w}$, we remove i from $setOfNeighbors$ at the end of the loop. For ease of presentation, we have excluded this operation from the outline of the reducer in Figure 14.

For the same reason, we have simplified the use of the function $getWeight()$ in Lines 10 and 16. In practice, its arguments depend on the selected weighting scheme:

- For CBS, it simply needs the array of frequencies as input, since w_{ij}=$frequencies[j]$.
- For ECBS and JS, it additionally requires the number of blocks containing e_i and e_j. This information is provided by an array that contains the number of blocks for all input entities. Due to its small size, this array can be loaded in memory in all available nodes.
- For EJS, $getWeight()$ additionally requires the node degree corresponding to every entity. This is equal to the number of non-redundant comparisons involving every entity and is computed through an additional MapReduce job. This job has almost the same functionality as Figure 14, but its `reduce` function stops at Line 7, only emitting the size of the set of neighbors for each entity (without counting the frequencies).
- For ARCS, $getWeight()$ requires only the cardinality of the blocks shared by every pair of entities, and not the frequency of their co-occurrence. Given that the reducer receives a list of whole blocks in its input value, the cardinality of each such block and the weight of each co-occurring entity can be directly computed in the first for loop (starting at Line 4). The rest of the process remains the same.

7. Stage 3: Cardinality Node Pruning (CNP)

Similar to WNP, CNP operates locally, retaining the best edges inside the neighborhood of every node/entity. Instead of a weight threshold, though, CNP uses a cardinality one, which retains the top k weighted edges within every node neighborhood. For its parallelization, we present three strategies, each

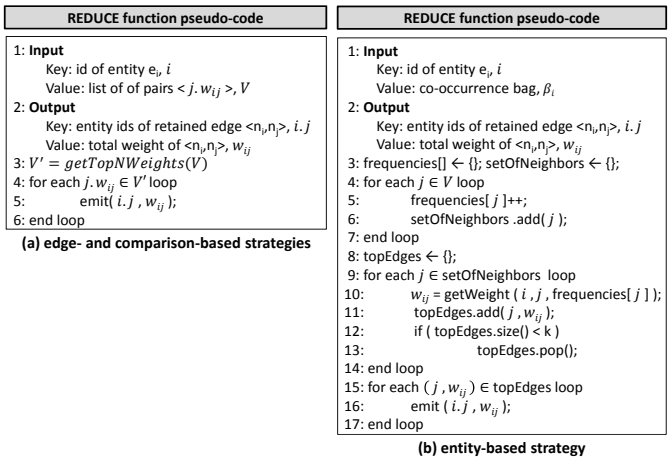


Figure 15: Pseudo-code interpretation of the reducers for CNP.

of which corresponds to the different output of the Preprocessing stage. They all require just one MapReduce job, whose mapper is identical with the respective one of WNP. For this reason, we exclusively focus on their reducers.

The edge-based strategy shares the same `reduce` function with the comparison-based one. Its functionality is outlined in Figure 15(a). It receives as input key the id i of the entity e_i that defines a neighborhood in the blocking graph G; its input value comprises the adjacent node/entity ids concatenated with the respective edge weights. In essence, the reducer orders all edges of the neighborhood in descending order of weight and extracts those placed in the top k ranking positions (Line 3). For each such edge, it emits the ids of its adjacent entities as key, $i.j$, along with its weight as value, w_{ij} (Lines 4-6).

In the example of Figure 13, the comparison-based strategy of CNP differs from WNP only in the reducer, which would emit only the top-k pairs of each group. Assuming that we want to retain the top-2 comparisons for every entity, out of the three comparisons shown in the group of e_1, we would emit $(e1.e3, 2/3)$ and $(e1.e2, 1/3)$.

For the entity-based strategy, the reducer appears in Figure 15(b). Its input key contains the entity id of the neighborhood center e_i, while its input value comprises the corresponding co-occurrence bag β_i. Similarly to WNP, it prepares the estimation of weights using two data structures: $frequencies$, the array of co-occurrence frequencies, and $setOfNeighbors$, the set of distinct neighbors (Lines 3-7). For every co-occurring entity, it estimates the corresponding weight and adds it in a sorted stack, $topEdges$ (Lines 8-11). If the size of the stack exceeds the cardinality threshold k, it removes the last item, i.e., the edge with the lowest weight (Lines 12-13). At the end of this loop, every edge that remains in the sorted stack is emitted as output; the concatenated entity ids form the key, i.e., $i.j$, while the value comprises its weight, i.e., w_{ij} (Lines 15-17).

8. Stage 3: Weighted Edge Pruning (WEP)

WEP estimates the average edge weight across the entire blocking graph and discards those edges that do not exceed it. Again, we propose three different parallelization approaches

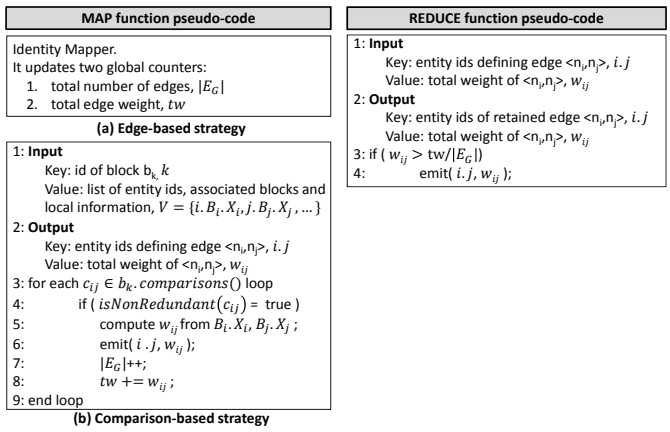


Figure 16: Pseudo-code interpretation of (a) the edge-based and (b) the comparison-based strategy for WEP.

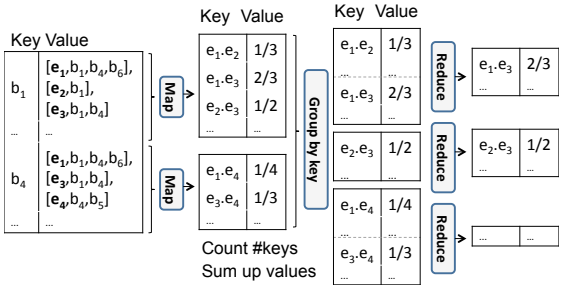


Figure 17: An example of the comparison-based strategy for WEP, using the JS weighting scheme.

that correspond to the three strategies of Preprocessing. The first two involve a single MapReduce job, while the third one requires two jobs.

8.1. Edge-based Strategy

Figure 16 presents the edge- and the comparison-based strategies for WEP. Once more, they share the same reducer, differing only in the mapper, which is determined by the preprocessing approach.

Figure 16(a) illustrates the `map` function of the edge-based strategy. It constitutes an identity mapper that receives the id $i.j$ of an individual edge $<n_i, n_j>$ as key and its weight w_{ij} as value. Before forwarding the input to the reducer, it updates two counters that are necessary for estimating the average edge weight: the size of the blocking graph $|E_G|$ (i.e., the number of its edges) and the total edge weight tw.

The `reduce` function receives the id and the weight of an individual edge, i.e., a pair $(i.j, w_{ij})$. If the weight is greater than the mean weight of the graph ($w_{ij} > tw/|E_G|$), the edge is retained and, thus, the input pair is emitted as output.

8.2. Comparison-based Strategy

The `map` function of this approach appears in Figure 16(b). It iterates over all comparisons in b_k and assesses the LeCoBI condition for the involved entities (Lines 3-4). For every non-redundant comparison c_{ij}, it estimates the corresponding edge weight w_{ij} from the information in the input value and emits it along with the edge id: `key`=$i.j$ and `value`=w_{ij} (Lines 5-6). It

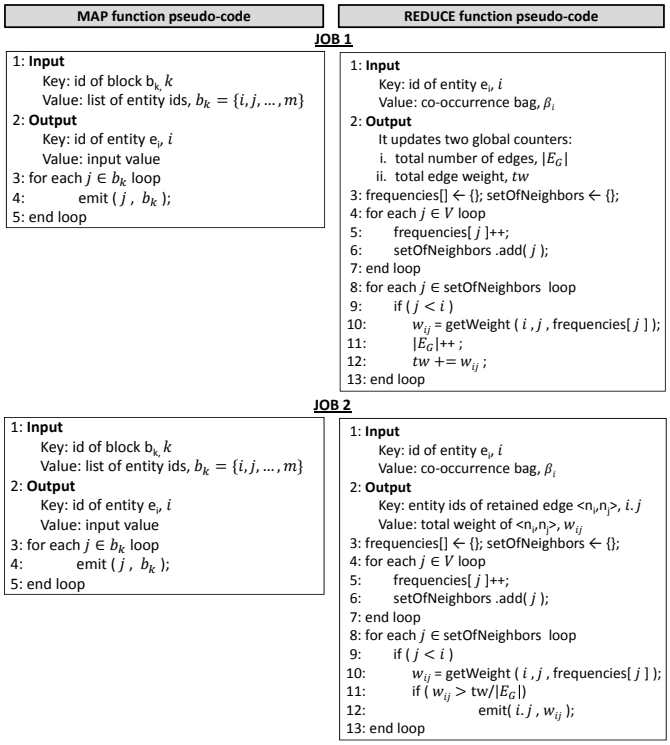


Figure 18: Pseudo-code interpretation of the entity-based strategy for WEP.

also updates the two counters that are used in the reduce phase, $|E_G|$ and tw (Lines 7-8).

The `reduce` function is the same as in the previous strategy.

In Figure 17, we present the functionality of the comparison-based strategy of WEP in combination with JS, when applied to the output of Figure 9. The `map` function receives the block $b_1 = \{e_1, e_2, e_3\}$ as input and for every non-redundant comparison in b_1, it outputs the id of the comparison as `key` (i.e., $e_1.e_2$, $e_1.e_3$ and $e_2.e_3$) and the corresponding weight as `value`. According to the JS weighting scheme, e_1-e_2 weight is 1/3, as the entities e_1, e_2 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.

8.3. Entity-based Strategy

This strategy involves two jobs that appear in Figure 18: the first one estimates the global weight threshold, which equals the average edge weight of the entire blocking graph, while the second one discards the edges that do not exceed this threshold. Unlike the previous strategies, these two procedures cannot be carried out in a single job, due to the absence of any auxiliary computations in Preprocessing. Thus, the edge weights and the weight threshold can only be estimated by the first reducer, while a second job is required in order to apply the threshold to the blocking graph.

The `map` function of the first job takes as input key the id k of an individual block b_k and as input value the entity ids it contains. For every entity $e_i \in b_k$, it emits a pair (`key`,`value`)=(i, b_k) as output. In this way, the reducer aggregates in its input value the co-occurrence bag β_i of the entity e_i that is given as input key. The goal is to estimate the weights of the incident edges and to derive the global weight threshold. Again, two global counters are employed in this process: the number of edges $|E_G|$ and the sum of all edge weights tw.

In more detail, the `reduce` function applies the standard procedure of the entity-based strategy, using two data structures: the set of distinct neighboring entities and the array recording their frequency of co-occurrence. In Lines 4-7, the reducer iterates once over the co-occurrence bag and updates the data structures accordingly. Then, it iterates over the distinct neighboring entities and estimates the corresponding edge weight for those having a lower entity id (Lines 8-10); this is done in order to avoid considering the weight of every edge twice – once in the reducer corresponding to each of its adjacent entities. Finally, it updates the two global counters (Lines 11-12).

To avoid a high I/O, the estimated edge weights are not saved on disk. Instead, they are recomputed and pruned in the second job. As a result, the second `map` function is identical with the first one, while the second `reduce` function aggregates the co-occurrence bag β_i of an individual entity e_i, too. This time its goal is to prune the incident edges with a weight lower than the global threshold. The first loop is identical with that of the first reducer, adding the necessary information to the two data structures (Lines 3-7). The second loop estimates again the weight w_{ij} of every edge such that $j < i$ (Lines 8-10). Then, it compares w_{ij} with the global weight threshold, $tw/|E_G|$; in case w_{ij} exceeds it, the edge $<n_i, n_j>$ is retained, emitting its id as output key and its weight as output value (Lines 11-12).

9. Stage 3: Cardinality Edge Pruning (CEP)

CEP retains the K edges with the largest weights across the entire blocking graph. Theoretically, this operation can be implemented in a job with a single reducer that gathers all edges and sorts them in descending weight. In practice, however, this approach does not scale to large blocking graphs with millions of nodes and billions of edges: there is no sufficient memory that can accommodate so many edges in a single node – even if it is equipped with tens of Gigabytes of RAM. To overcome this limitation, the main idea behind all strategies is to convert the global cardinality threshold into a global weight one.

Again, we present three different strategies that correspond to the three Preprocessing approaches. The edge- and comparison-based ones involve two MapReduce jobs: the first one identifies the minimum edge weight w_{min}, such that at least K edges have a weight greater than or equal to it; the second job outputs exactly K edges with a weight greater than or equal to w_{min}. The entity-based approach involves an additional job that derives the edge weights from the input blocks.

9.1. Edge-based Strategy

The two jobs of this approach appear in Figure 19 together with those of the comparison-based strategy, which shares the same reducer for every job.

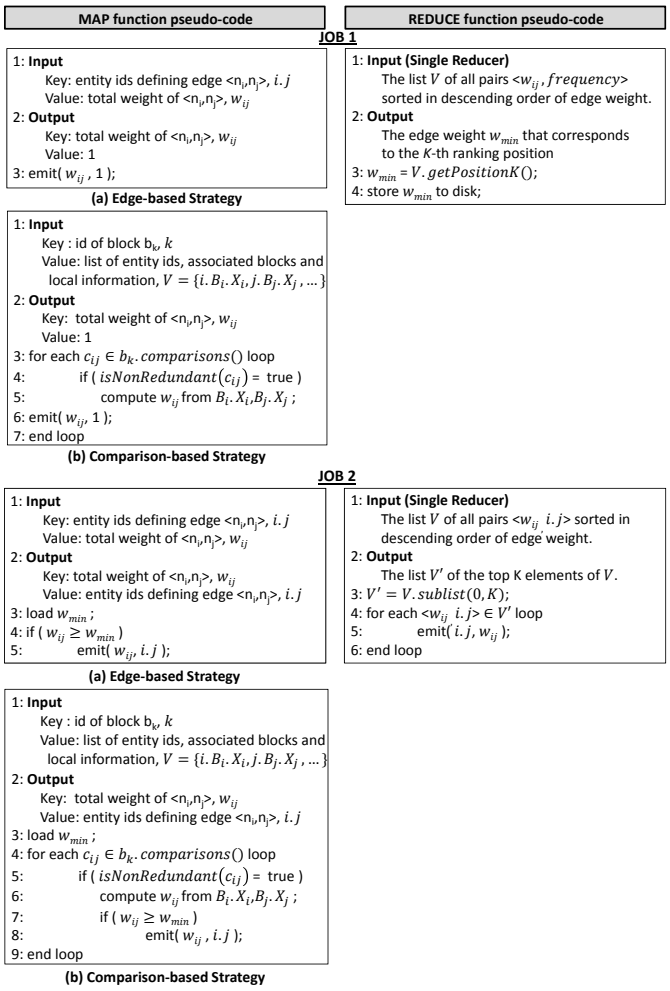


Figure 19: Pseudo-code interpretation of (a) the edge-based and (b) the comparison-based strategy for CEP.

The first `map` function receives as input key the id $i.j$ of an individual edge $<n_i, n_j>$ and as input value the corresponding weight w_{ij}. As output, it emits a pair $(w_{ij}, 1)$, thus enabling the single reducer of this job to count the number of edges with the same weight.

Indeed, the first `reduce` function receives as input the list of all distinct weights, sorted in descending order, along with their frequencies. It goes through this list starting from the top (i.e., the largest weight) and keeps a counter with the number of edges that have a weight greater than or equal to the current weight. As soon as the counter reaches K, the reducer stops and stores the corresponding weight, w_{min}, to the disk.

The second `map` function of this strategy has the same input as the first one: `key`=$i.j$, `value`=w_{ij}. It loads w_{min} from the disk and compares it with w_{ij}. If $w_{ij} \geq w_{min}$, it swaps key and value, emitting a pair $(w_{ij}, i.j)$.

The second `reduce` function receives these pairs and sorts them in descending order, from the largest weight to the lowest one. The reducer extracts exactly top K elements, thus addressing the ties that may arise, i.e., the situation where the overall number of edges $<n_i, n_j>$ with $w_{ij} \geq w_{min}$, is larger than K. Finally, it emits the retained pairs by swapping keys and values, i.e., `key`=$i.j$, `value`=w_{ij}.

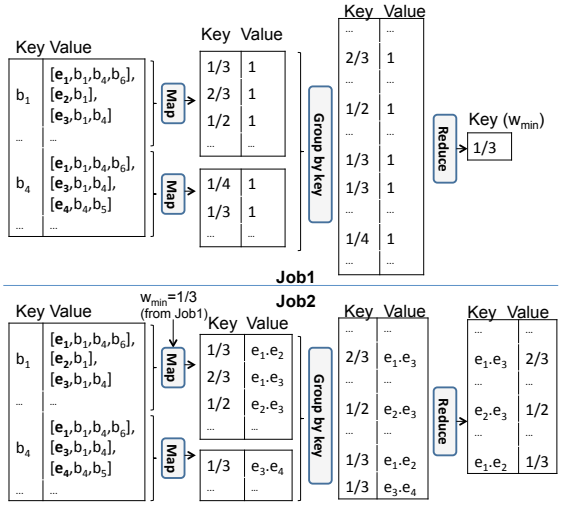


Figure 20: An example of the comparison-based strategy for CEP, using the JS weighting scheme.

9.2. Comparison-based Strategy

As mentioned above, this strategy differs from the edge-based one only in the functionality of the mappers, which are outlined in Figure 19. We observe that for every job, the `map` function operates on the enriched description of an individual block, iterating over all its comparisons. For every non-redundant comparison between e_i and e_j, the mapper computes the corresponding edge weight w_{ij} from the associated block ids and local information. As output, the first `map` function emits a pair $(w_{ij}, 1)$, while the second one emits a pair $(w_{ij}, i.j)$, only if $w_{ij} \geq w_{min}$.

In Figure 20, we illustrate the functionality of the comparison-based strategy of CEP in combination with JS, when applied to the output of Figure 9. 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 comparisons 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. For the `reduce` function, the input comprises all retained comparisons, sorted in descending order of weight; starting from the top of this list, we emit each pair until we reach the k-th pair, which is e_1-e_2 (pairs with the same weight are sorted randomly).

9.3. Entity-based Strategy

The three jobs comprising this strategy appear in Figure 21. The first one estimates the edge weights, the second sorts them in descending weight in order to identify w_{min}, and the third retains the edges with weight higher than or equal to w_{min}.

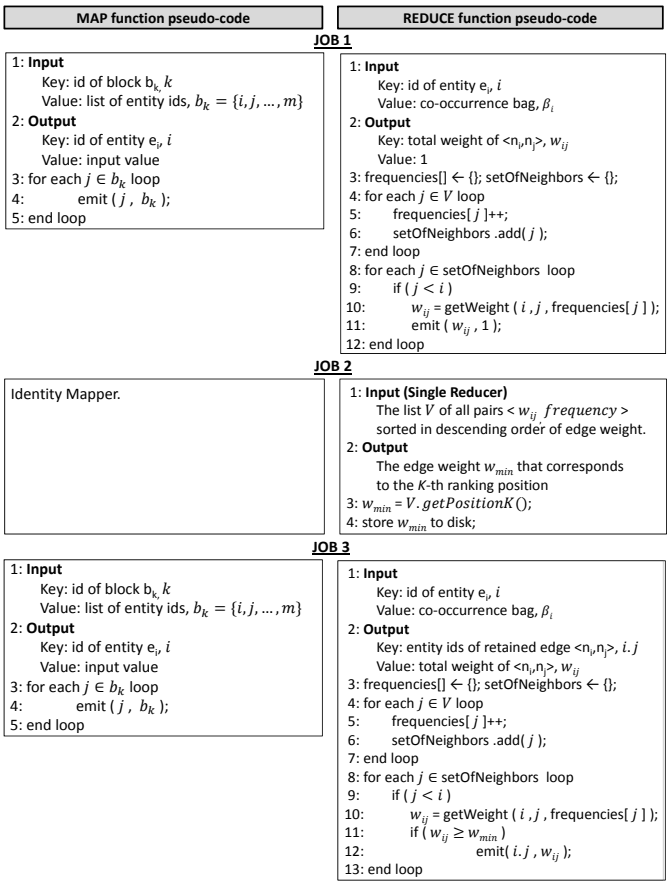


Figure 21: Pseudo-code interpretation of the entity-based strategy for CEP.

In more detail, the `map` function of the first job receives as input key the id k of an individual block b_k and as input value the entity ids it contains. For every entity $e_i \in b_k$, it emits its id i as output key and the entire block as output value.

As a result, the first reducer gathers in its input value the co-occurrence bag β_i of the entity e_i, which is assigned to the input key. Then, it estimates in Lines 3-10 the weights of all edges $<n_i, n_j>$ such that $j < i$. Following the typical methodology of this strategy, it uses an array that records the frequency of co-occurrence for every entity connected with e_i (Line 5) and gathers the distinct neighbors of e_i in a set (Line 6). In the end, it emits a pair $(w_{ij}, 1)$ for every estimated edge weight.

The second job involves an identity mapper that forwards all edge weights to a single reducer. The weights are automatically sorted from the largest to the smallest one and every distinct weight is associated with its frequency. The reducer simply iterates over the sorted list of weights and retrieves the one corresponding to the K-th largest value. This weight, denoted by w_{min}, is then stored on disk.

The third job recomputes all edge weights with the aim of retaining those higher than or equal to w_{min}. Thus, it shares the same `map` and `reduce` functions with the first job. The only difference is in the output of reducer: in Lines 11-12, it ensures that only weights exceeding w_{min} are retained, emitting the concatenated entity ids as output key and their weight as output value, i.e., `key`=$i.j$, `value`=w_{ij}.

Algorithm 1: MaxBlock

Input: B the current block collection
Output: P the set of block partitions

1 $B' \leftarrow$ sort(B); `// sort in descending cardinality`
2 $b_0 \leftarrow B'$.remove(0); `// remove largest block`
3 $maxCost \leftarrow \|b_0\|$; `// max comparisons per partition`
4 $P_0 \leftarrow \{b_0\}$; `// first partition`
5 $Q \leftarrow \{P_0\}$; `// priority queue, sorting partitions in ascending cost`
6 **while** $B' \neq \{\}$ **do** `// while not empty`
7 $b_i \leftarrow B'$.remove(0); `// remove first block`
8 $P_{head} \leftarrow Q$.poll(); `// get lowest cost partition`
9 $totalCost \leftarrow \|b_i\| + P_{head}$.currentCost();
10 **if** $totalCost \leq maxCost$ **then**
11 $P_{head} \leftarrow P_{head} \cup \{b_i\}$; `// add to partition`
12 **else**
13 $P_i \leftarrow \{b_i\}$; `// create new partition`
14 Q.add(P_i); `// add to queue`
15 Q.add(P_{head}); `// place back to queue`
16 **if** $B' = \{\}$ **then** `// if all blocks were processed`
17 $P_{head} \leftarrow Q$.poll(); `// get smallest partition`
18 **if** *isRemnantCluster*(P_{head}) = `true` **then**
19 $B' \leftarrow B' \cup P_{head}$; `// re-process its blocks`
20 $maxCost \leftarrow maxCost + P_{head}$.currentCost() / $|Q|$;
21 **else**
22 Q.add(P_{head});
23 **return** Q;

10. Load Balancing

A typical bottleneck in MapReduce algorithms is the unbalanced workload that is assigned to the map or reduce tasks in each MapReduce job. In practice, data follow a skewed distribution, which results in groups of data being significantly larger than others. The map or reduce tasks that process these larger groups need substantially more time to finish, determining the efficiency of the whole job. Load balancing, indeed, affects both phases of the MapReduce job, as the reduce phase cannot start processing the output of the map phase, until all map tasks have finished, and the job is not finished unless all reduce tasks are completed.

10.1. Default Load Balancing

The default load balancing implementation of Hadoop is a hash-based algorithm, which assigns each group of data, determined by the output key of the map phase, to a bucket in a hash table. The buckets correspond to data partitions, each of which is input to a distinct reduce task. Consequently, the number of data partitions is equal to the number of reduce tasks. Notice that a reduce task can be assigned to more than one keys, since a bucket in a hash table corresponds to more than one hashed keys. This means that the default hashing-based load balancer of Hadoop, given a good hash function, can achieve a very good distribution in the number of keys that each partition (reduce task) will receive. However, in skewed data, this does not guarantee a balanced workload.

For example, assume that we have $p = 10$ partitions, $i = 100$ distinct keys $k_1, ..., k_i$, each corresponding to a word, ordered in descending frequency, and each key k_i has $|k_i|$ values, where the distribution of $|k_i|$ abides by Zipf's law. Assuming we have $1,000$ values in total, i.e., $\sum |k_i| = 1,000$, then $|k_1| = 1,000 \cdot \frac{1}{\sum_i 1/i} \approx 192$, $|k_2| = |k_1|/2 \approx 96$, ..., $|k_{100}| = |k_1|/100 \approx 2$. The default balancer of Hadoop would ideally assign $i/p = 10$ keys per partition. Thus, the partition that will receive the most frequent key k_1, associated with $1/5$ of the total values, will also receive 9 more keys, and it is highly likely that this partition will be one of the slowest to process.

In the case of entity resolution algorithms, the imbalance of the workload is even greater, as the keys typically correspond to block ids, and the values correspond to entities in those blocks, which have to be compared. Hence, the workload of each reduce task is quadratic to the number of input values it receives. In the previous example, the total number of comparisons would be $\sum |k_i| \cdot (|k_i| - 1)/2 \approx 64,500$, while the biggest block k_1 would yield $18,336$ comparisons, i.e., approximately $1/3$ of the total comparisons.

10.2. MaxBlock Load Balancing

To address this issue, we developed a specialized algorithm for load balancing, named *MaxBlock*. Our goal is to split the input blocks into partitions with a balanced number of comparisons. In order to ensure better results, the number of partitions is determined dynamically. Intuitively, our load balancing strategy is to assign the biggest block to a partition of its own and set the number of comparisons in this partition as the upper threshold of comparisons for every other partition. Then, we create a new partition and keep adding blocks to this new partition, until this threshold is reached. When the threshold is reached, we create a new partition, and continue this process until all blocks have been assigned to a partition.

The functionality of *MaxBlock* is outlined in Algorithm 1. It sorts the block collection in descending cardinality (Line 1) and removes the first and largest block, b_0 (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 a priority queue Q, keeping partitions in ascending order of comparisons (Line 5). 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 back to Q (Line 15).

As we demonstrate in the experimental evaluation, all partitions share practically the same computational cost, except for the smallest one, which merely covers a small fraction of *maxCost*. Yet, it contains the vast majority of the blocks, with each one involving a handful of comparisons. This is called *remnant cluster* and it corresponds to the tail of the power-law distribution of block cardinalities. To achieve a perfectly flat distribution of costs, our algorithm distributes the blocks of the remnant cluster to the other partitions.

This functionality is performed by the second `if` statement in Algorithm 1. Line 16 checks whether all blocks have been placed into a partition, thus terminating the first iteration. Lines 17 and 18 examine whether the smallest partition is a remnant cluster, i.e., whether it contains more than 50% of all blocks and their total computational cost is lower than 90% of *maxCost*. In case both conditions are satisfied, the blocks of the remnant cluster are put back into the processing list (Line 19); in addition, *maxCost* is updated so that their computational cost is evenly split among the other partitions (Line 20). In case of a negative check, the smallest partition is placed back into the priority queue and the process is terminated.

On the whole, the time complexity of *MaxBlock* is determined by the sorting of blocks and the use of the priority queue, whose operations cost $O(\log |B|)$ per block. Therefore, the overall time complexity is $O(|B| \log |B|)$, which means that *MaxBlock* scales well to large block collections, involving a negligible overhead, as shown in Section 11. For example, the actual cost of sorting the blocks is the cost for sorting up a few million of integer values, each representing a block cardinality. This operation does not require more than a few seconds.

10.3. MaxBlock Implementation

The results of *MaxBlock* are fed to the Partitioner class, which is responsible for assigning a reduce task to each key. We have overridden the default Partitioner to just send each key to the partition that *MaxBlock* has defined for this key. This approach is primarily used to balance the functions with quadratic time or space complexity. The former case involves functions that iterate over all comparisons in a block. For the edge-based strategy, this is the `reduce` function in the first job for Stage 2 - Preprocessing (Figure 6). For the comparison-based strategy, this case applies to all `map` functions for Stage 3 - Metablocking (Figures 12(b), 16(b) and 19(b)).

Quadratic space complexity appears in the case of the entity-based strategy, where the bottleneck is the I/O overhead of its `map` function in Stage 3: the size of its output is quadratic with respect to number of entities in the input value, since the whole block is emitted for each of the contained entities. As a result, most mappers have to write only a few intermediate key-value pairs, while those that deal with the bigger blocks have to emit a much larger bulk of data. To address this issue, we use *MaxBlock* to balance the output of entity-based Preprocessing (Figure 11). Our goal is to split the blocks into partitions with equal size of representation in bytes. This can be easily done by redefining the cost of a block as the number of bytes that are required for the compressed representation of its entities (i.e., after sorting them in ascending id and replacing every id by its difference with the previous one).

11. Experiments

The goal of our experimental analysis is threefold: *(i)* to demonstrate that our approaches scale well to large block collections stemming from Web data, *(ii)* to compare the relative efficiency of the edge-, comparison- and entity-based strategies,

	$D_{dbpedia}$		$D_{freebase}$	
	D_1	D_2	D_1	D_2
Entities $\|E\|$	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 $\|D(E)\|$	892,579		1,347,266	
BF Comparisons	$2.58{\cdot}10^{12}$		$1.33{\cdot}10^{13}$	

Table 2: The large, heterogeneous entity collections that were 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
$\|B\|$	1,239,424	1,499,534	1,309,145	4,522,222
$\|\|B\|\|$	$4.23{\cdot}10^{10}$	$8.00{\cdot}10^{10}$	$1.05{\cdot}10^{11}$	$2.19{\cdot}10^{11}$
$\|D(B)\|$	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 4: The block collections that were given as input to Meta-blocking.

and *(iii)* to assess the relative efficiency of the various Meta-blocking configurations.

We begin with the setup of our experimental analysis in Section 11.1. In Section 11.2, we present a comparison between the default balancer and *MaxBlock*. In Section 11.3, we show the time efficiency of all strategies for the four pruning schemes in combination with the five weighting schemes; we also discuss their relative efficiency in view of a similar comparison in the case of the serialized workflow. Section 11.4 analyzes the scalability of the comparison-based and entity-based strategies, while Section 11.5 elaborates on the qualitative performance of the Meta-blocking techniques. We conclude with a discussion on the findings of our experiments in Section 11.6.

11.1. Setup

All approaches were implemented in Java, version 7, using Apache Hadoop, version 1.2.0. They are compatible with more advanced frameworks, such as Apache Spark[7] and Apache Flink[8], but Hadoop is a more established framework and suits well to the goals of our experimental analysis.

All experiments were performed on a cluster[9] 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 amounted to 4 TB and was equally partitioned among the 15 nodes. For Load Balancing, we employed the default mechanism of Hadoop for the `map` and `reduce` functions that involve a processing of linear complexity. For those involving a quadratic complexity, we distributed the relevant blocks to the available nodes using *MaxBlock*, as explained in Section 10.3.

Datasets. To evaluate the performance of our approaches, we employ the largest datasets that have been applied to Meta-blocking. Their technical characteristics appear in Table 2.

$D_{dbpedia}$ involves entities stemming from two snapshots of the DBpedia[10] Infoboxes in English, which chronologically differ by 2 years – D_1 corresponds to version 3.0rc and D_2 to version 3.4. In total, they comprise 3.3 million entities, of which less than 900,000 are common (i.e., they have the same URL). This dataset has been previously employed in the literature [4, 6, 11, 15, 16]. The second dataset, $D_{freebase}$, contains entities from the Billion Triple Challenge 2012[11]. In this case, D_1 encompasses the entities from DBpedia and D_2 the entities from Freebase[12]. For both collections, we have disregarded all URIs that appear in just one triple so as to avoid noisy entity profiles. In total, there are 7.4 million entities, of which 1.3 million are common according to the *owl:sameAs* statements.

Given that both datasets comprise two individually clean (i.e., duplicate-free) entity collections, D_1 and D_2, they are inherently suitable for *Clean-Clean ER*; the goal in this task is to identify the matching entities shared by D_1 and D_2, without performing any comparisons inside the individual collections. In our experiments, we use both datasets for *Dirty ER*, as well, by merging D_1 and D_2 into a single dirty collection that contains matches in itself. The goal in this task is to partition the entity collection into clusters of matching entities. The ground truth is provided by existing owl:sameAs links between Freebase and DBpedia for $D_{freebase}$. Since $D_{dbpedia}$, involves two different snapshots of DBpedia, we consider matching profiles those having the same subject URI.

We used Token Blocking [6, 11] in order to derive redundancy-positive block collections from the entity profiles of the two datasets. We also applied Block Purging [11] to the original blocks in order to discard the extremely large ones that contain almost half the input entities. The technical characteristics of the resulting blocks appear in Table 4. In total, we have four block collections, two for each ER task, that vary significantly in their characteristics, for example, in the number of blocks per entity (BPE).

Measures. To assess the *effectiveness* of the (restructured) block collections, we employ four established measures [2, 3, 10, 11, 17]:

(i) Pairs Completeness (PC) expresses recall – see Section 2.

(ii) Pairs Quality (PQ) estimates precision – see Section 2.

(iii) Reduction Ratio (RR) estimates the relative decrease in the number of comparisons conveyed by Meta-blocking. It is formally defined as: $RR = 1 - \|B'\|/\|B\|$, where B is the original and B' the restructured block collection.

(iv) $H3R$ expresses the harmonic mean of PC and RR, i.e., $H3R = (2 \cdot PC \cdot RR)/(PC + RR)$. It provides an intuitive way to measure the trade-off of (Meta-)blocking, quantifying the extent to which a blocking method manages to significantly reduce the number of required comparisons, without missing many matches.

[7] `https://spark.apache.org`

[8] `https://flink.apache.org`

[9] provided by GRNET's ~okeanos (`https://okeanos.grnet.gr`)

[10] `http://dbpedia.org`

[11] `https://km.aifb.kit.edu/projects/btc-2012`

[12] `https://www.freebase.com`

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

Table 3: The distribution of partition cardinalities produced by the default load balancer of Hadoop, PairRange and *MaxBlock*.

All effectiveness measures are defined in the interval $[0, 1]$, with higher values indicating higher effectiveness.

To assess the *time efficiency* of (Meta-)blocking methods, 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 method.

11.2. Load Balancing

In this section, we examine the performance of Load Balancing with respect to the computationally most intensive functions of the three strategies for parallel Meta-blocking, i.e., the functions with quadratic time or space complexity.

Remember that quadratic time complexity appears in the `reduce` function of the first Preprocessing job for the edge-based strategy (see Figure 6) as well as in all `map` functions of Stage 3 for the comparison-based strategy (see Figures 12(b), 16(b) and 19(b)). All these functions iterate over all comparisons in the input blocks in order to estimate the corresponding edge weights. As a result, Load Balancing aims to split the original block collection into disjoint partitions with (ideally) the same *partition cardinality*, i.e., the same total number of comparisons in the blocks of the partition; every partition is then assigned to one of the available nodes for its processing.

We compare the performance of *MaxBlock* with two baseline methods: the default balancer of Hadoop and *PairRange*, the state-of-the-art Load Balancing algorithm that was originally introduced in [18]. In essence, PairRange splits evenly the comparisons of a block collection into a predefined number of partitions, by assigning every comparison to a particular partition id. To this end, it involves a single MapReduce job, whose mapper associates every entity e_i in block b_k with the output key $rid.k.i$, where rid denotes the index of the comparison range, i.e., the partition id. Then, the reducer groups together all entities that have the same rid and block id, reproducing the comparisons corresponding to partition rid.

To compare the two baseline methods with *MaxBlock*, we consider the distribution of the partition cardinalities they produce. We actually summarize these distributions through their minimum, maximum, median and mean partition cardinalities. The closer these measures are to each other, the more balanced is the workload assigned to each node. We applied all approaches to the input of Stage 2 of the parallelized workflow, i.e., after applying Block Filtering to the original block collections (see Figure 3(b)). The outcomes of our experiments appear in Table 3.

Note that PairRange receives the number of ranges (partitions) as input from the user. This requires the user to manually inspect the data at hand, which is cumbersome. In our experiments, we gave PairRange an unfair advantage by using the same number of ranges as those in MaxBlock. As a result, we observe that the two algorithms produce identical sets of partitions. Most importantly, though, their partitions exhibit a practically constant distribution of cardinalities across all datasets: all 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. For DB_C and DB_D, it yields a normal distribution, as the median and the average cardinalities almost coincide, lying close to the middle of the maximum and the minimum ones. The standard deviation is an order of magnitude lower than the other measures, thus indicating minor differences in the computational cost of the various partitions. However, the performance of the default balancer aggravates in the case of FR_C and FR_D, where the standard deviation is almost equal to the average cardinality. Another indication is that the difference between the minimum and the maximum cardinality raises to 4 and 1 orders of magnitude, respectively. Their medians suggest that the distribution of FR_C is dominated by partitions smaller than the mean cardinality, and vice versa for FR_D.

These patterns indicate that serious bottlenecks are expected to rise in the case of the default load balancer of Hadoop. For this reason, we did not measure the actual running time it yields. Neither do we present the running time of PairRange. The reason is that it is almost identical with that of *MaxBlock*, which appears below, in Section 11.3. In fact, PairRange is slower than *MaxBlock* by a couple of minutes, due to the higher overhead it involves: to adapt it to the functions of quadratic time complexity, an additional MapReduce job is required for both the edge- and the comparison-based strategy.

In more detail, we can integrate PairRange into the edge-based strategy by modifying the first `reduce` function of Figure 6 so that a global counter estimates the total number of com-

parisons, while the input is emitted without any further processing. Thus, the `map` function of a second, new job receives as input an individual block and applies the mapper of PairRange to it. The second reducer receives a balanced comparison range as input and estimates the corresponding edge weights (i.e., it applies the `reduce` function of Job 1 in Figure 6). Finally, the third job applies Job 2 of Figure 6 without any modifications.

For the comparison-based strategy, PairRange needs to extend the `reduce` function of Preprocessing in Figure 8 so that it estimates the total number of comparisons. Then, we need to add a new MapReduce job to every pruning algorithm; the `map` function receives individual blocks and applies the mapper of PairRange to them, while the `reduce` function receives a balanced comparison range as input and applies the functionality of the `map` functions in Figures 12(b), 16(b) and 19(b), i.e., it estimates the corresponding edge weights. Finally, a second MapReduce job is required for every pruning algorithm; it consists of an identity mapper and the `reduce` functions in Figures 12(b), 16(b) and 19(b).

Regarding the entity-based strategy, the goal of Load Balancing is to address the quadratic space complexity that appears in all `map` functions of Stage 3 (see Figures 14, 18 and 21). This can be achieved by balancing the output of Preprocessing in Figure 11. Yet, among the three load balancing mechanisms, only the default one provided by Hadoop applies to this task without any modifications. As explained in Section 10.3, *MaxBlock* needs to adopt a new cost function, which expresses the disk space that is occupied by the compressed representation of the entities contained in every block.

However, PairRange cannot consider alternate cost functions, as it is inherently crafted for balancing comparisons. To adapt it to the entity-based strategy, we need to modify its functionality so that every block is entirely contained in a single comparison range (partition). In other words, we need to ensure that the comparisons of no block are spread across multiple partitions; otherwise, we have to alter the functionality of the entity-based mappers of Stage 3, which is out of the scope of this evaluation. To meet this requirement, the number of comparison ranges should be equal to or less than those of *MaxBlock*. We actually consider two configurations: using the same number of partitions as *MaxBlock* (PairRangeI) and using half the partitions of *MaxBlock* (PairRangeII). Note that PairRangeI does not necessarily produce the same distributions as PairRange in Table 3, because some blocks are larger than the remaining space in their partition, but are not broken into smaller chunks.

To evaluate the performance of Load Balancing for the entity-based strategy, we do not consider the distribution of comparisons among partitions. Instead, we are more interested in the compressed representation of blocks in bytes and the corresponding I/O overhead. We indirectly evaluate this aspect through the overhead time of all entity-based pruning algorithms. Table 5 presents the corresponding performance on top of DB_C, using the *CBS* weighting scheme for each algorithm. The rest of the datasets and weighting schemes yield similar results and are omitted for brevity.

We observe that *MaxBlock* exhibits the highest overhead, compared to the default balancer, due to its cost function, which compresses the representations of blocks before clustering them into partitions. PairRangeII is faster than PairRangeI, due to the lower number of partitions it involves, while the default balancer has 0 overhead, as it is the baseline of the overhead of the load balancing algorithms. Regarding the overall time, we observe that *MaxBlock* consistently provides the best execution times, with the default balancer being the least efficient one in most cases: it yields slower times than *MaxBlock* by 12% (CEP) to 20% (WEP). The two variations of PairRange fluctuate between these two extremes, with PairRangeI being consistently more efficient, because the larger number of partitions it employs ensures a more balanced I/O overhead across the nodes. Note that PairRangeII appears to be less efficient than the default load balancer over WNP, but their difference should be attributed to its execution overhead.

	Overhead	CNP	WNP	CEP	WEP
Default	0	88	73	163	147
PairRangeI	2	77	68	145	130
PairRangeII	1	83	74	156	139
MaxBlock	3	74	65	145	123

Table 5: The wall-clock time (in minutes) of Meta-blocking using the default Hadoop balancer, the two variations of PairRange, and MaxBlock for the entity-based strategy over DB_C, using the *CBS* weighting scheme across all pruning algorithms. The overhead of executing each load balancing algorithm, compared to the default balancing, is common for all pruning algorithms and is included in the wall-clock times.

On the whole, we conclude that *MaxBlock* consistently outperforms the default mechanism of Hadoop across all parallelization strategies and pruning algorithms, even if it comes with a small execution overhead, compared to the default balancer. Moreover, *MaxBlock* is scalable – $O(|B| \cdot \log |B|)$ – and terminates within a few minutes for all datasets, as shown in Table 5.

Compared to PairRange, *MaxBlock* has three advantages: *(i)* It determines the number of partitions automatically, through a data-driven procedure. Instead, PairRange receives this parameter as input, requiring the user to specify it, after manually inspecting the data at hand. *(ii)* For the edge- and comparison-based parallelization strategies, *MaxBlock* consistently yields lower overall execution times than PairRange, as it saves a whole MapReduce job. *(iii) MaxBlock* is more flexible and generic than PairRange. Thus, it can be easily adapted to the entity-based strategy, by incorporating a cost function that tackles quadratic space complexity. Instead, PairRange is only suitable for balancing functions that suffer from quadratic time complexity, due to the number of comparisons they process.

11.3. Time Efficiency

We applied the three parallelization strategies of all Meta-blocking techniques to the four datasets 2 times and measured the corresponding average Overhead Time. The outcomes are presented in Table 7[13]. Note that the edge-based strategy was

[13] Some of the times presented here are slightly improved, compared to [9], due to implementation improvements, e.g., using Java Arrays instead of Maps or Lists, when applicable.

	$D_{dbpedia}$		$D_{freebase}$	
	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 6: The block collections after Block Filtering.

inapplicable to FR_C and FR_D, as its space requirements exceeded the available 4 TB of disk space. For the other two datasets, we terminated prematurely the processes that ran for more than 6,000 minutes (100 hours), all of which were still far from completion. Below, we analyze the performance of each stage of the parallelized workflow of Meta-blocking.

Stage 1. The goal of this stage is to apply Block Filtering to the input block collection. In Table 7, we observe that the basic and the advanced strategy exhibit practically equivalent overhead times. Remember that the former involves two jobs that order once and globally the input blocks, whereas the advanced strategy entails a single job that sorts repeatedly and locally the input blocks. We can conclude, therefore, that the basic strategy offsets the cost of using two jobs by avoiding the computations that are repeated by the advanced one. However, the main reason for the equivalent overhead times is the linear time complexity of Block Filtering and its simple functionality that processes very large block collections at a negligible cost.

It should be stressed here that the exemplary performance of Block Filtering justifies the lack of a specialized load balancing algorithm for the functions with linear complexity.

Also worth noting is the qualitative performance of Block Filtering, which is presented in Table 6. We observe that despite its simple functionality, Block Filtering conveys significant enhancements in efficiency at a minor cost in recall. The total cardinality of all block collections is reduced by more than 60%, while their recall (PC) drops by less than 2%. As a result, the precision (PQ) raises by 3 times, on average. The number of blocks remains almost intact, but the average number of blocks per entity (BPE) is significantly reduced. In this way, the computation of edge weights is accelerated to a considerable extent.

Stages 2 & 3. To compare the parallelization strategies for Meta-blocking on an equal basis, Table 7 considers the performance of Stages 2 and 3 as a whole; note that the entity-based strategy was applied only to DB_C and DB_D, because its space requirements over the two larger datasets exceeded the available disk space (4 TB). Special care has been taken to highlight the relative efficiency not only of the three strategies, but also of the pruning and weighting schemes. For this reason, we examine these aspects separately in the following.

Parallelization Strategies. We observe that when moving from left to right in Table 7, i.e., from the smallest block collection to the largest one, the Overhead Time increases analogously for all parallelization strategies. Even for the largest dataset, though, most Meta-blocking methods require less than 2 days (~3,000 minutes), thus being much faster than the serial processing, which requires almost 8 days over the high-end server described in Section 1. Most importantly, though, there is a considerable discrepancy among the efficiency of the three parallelization strategies, which designates that the parallelization of Meta-blocking is not a trivial task.

In more detail, the edge-based strategy is consistently slower than the comparison-based one. Their difference is particularly intense in the case of edge-centric pruning schemes, but is significantly reduced for the node-centric ones. There are two exceptions that prove this rule: for CNP and WNP in combination with JS, the edge-based strategy is faster (by less than 3 minutes) than the comparison-based one over DB_D.

Regarding the entity-based strategy, it is significantly faster than the other strategies in the case of the node-centric pruning schemes across all datasets. For DB_C, for instance, it is 5 times faster than the comparison-based strategy of WNP in combination with all weighting schemes. Compared to the edge-based strategy, it is 9 times faster, on average, for the same pruning scheme and dataset. In the case of the edge-centric algorithms, though, the entity-based strategy outperforms only the edge-based one; compared to the comparison-based strategy, it requires at least twice as much time.

There are two factors that determine the relative performance of the parallelization strategies. The first one is the number of MapReduce jobs they involve. The larger this number is, the higher the overhead becomes and the less efficient is the corresponding strategy. This explains the inferior performance of the edge-based strategy, when compared to the comparison-based one: its Preprocessing involves one more job in order to calculate the weights of all edges in the blocking graph. The same holds for the entity-based strategy, when it is combined with the edge-centric pruning schemes; in this case, the entity-based strategy employs one more job than the comparison-based one in order to calculate the global pruning criterion in the absence of preprocessing computations in Stage 2.

The second important factor for the efficiency of the parallelization strategies is the I/O they involve between the independent nodes of the cluster. The higher the I/O of a strategy is, the higher is its overhead and the lower is its efficiency. Comparing the edge- and comparison-based strategies in this respect, the former involves a higher I/O, because it materializes an edge for every comparison in the input blocks – even the redundant ones. In contrast, the comparison-based approach creates a distinct edge only for the non-redundant comparisons. In our datasets, the latter approach yields around 30% less comparisons. An even more efficient approach is implemented by the entity-based strategy, which sends none of edges through the network. Instead, it exchanges the nodes of the blocking graph, as their number is typically orders of magnitude lower than the number of edges. By attaching the necessary information to every graph node, the edges can be created, weighted, and pruned locally, inside the independent nodes of the cluster.

Overall, we recommend using the entity-based strategy for node-centric pruning algorithms, and the comparison-based strategy for edge-centric ones.

Pruning Schemes. WEP is the most efficient method for the edge- and comparison-based strategies across all datasets, because it involves the simplest processing. For these strategies, the second fastest method is CEP, since it merely adds one job

		DB_C			DB_D			FR_C		FR_D	
Block Filtering	Basic	2			2			3		6	
	Advanced	2			2			3		6	
		Edge Based	Comp. Based	Entity Based	Edge Based	Comp. Based	Entity Based	Comp. Based	Entity Based	Comp. Based	Entity Based
CEP	ARCS	252	89	184	>6,000	135	431	1,319	1,244	3,359	3,065
	CBS	222	55	145	250	87	363	781	1,343	2,556	2,842
	ECBS	240	78	210	278	110	487	841	1,652	2,663	3,257
	JS	223	60	190	279	94	466	777	1,480	2,574	2,832
	EJS	1,996	116	>6,000	>6,000	180	>6,000	1,166	>6,000	4,090	>6,000
CNP	ARCS	554	370	73	>6,000	625	191	2,109	605	3,934	970
	CBS	491	301	74	559	527	186	1,488	643	2,514	995
	ECBS	555	383	76	639	633	197	1,949	665	3,058	1,187
	JS	534	363	83	618	620	210	1,637	656	2,546	977
	EJS	2,645	430	142	>6,000	733	382	2,319	1,069	5,222	1,993
WEP	ARCS	203	65	389	>6,000	99	319	520	1,006	1,802	1,967
	CBS	220	50	123	250	76	338	501	1,088	1,414	2,031
	ECBS	219	54	123	254	83	342	555	1,164	1,438	1,945
	JS	219	54	132	254	84	340	540	1,097	1,431	2,093
	EJS	1,993	81	204	>6,000	124	517	837	1,555	2,419	3,025
WNP	ARCS	562	363	63	>6,000	647	185	2,068	685	3,904	977
	CBS	498	304	65	569	539	196	1,534	541	2,671	1,313
	ECBS	568	389	73	658	647	193	1,971	588	3,046	1,238
	JS	553	373	74	641	644	202	1,636	690	2,790	1,176
	EJS	2,626	411	142	>6,000	700	379	2,317	1,041	5,214	2,211

Table 7: Overhead Time (*OTime*) in minutes for all Meta-blocking techniques across the four real datasets.

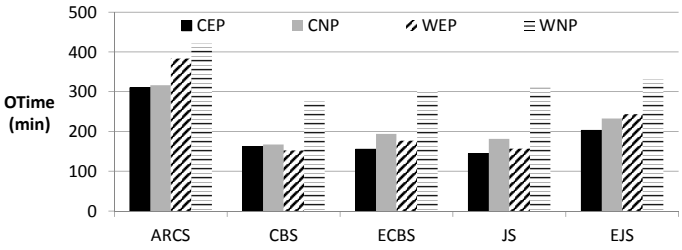


Figure 22: Overhead Time in minutes for all configurations of the serialized workflow over DB_C.

to the functionality of WEP in order to convert the cardinality pruning criterion into a weight one. For the entity-based strategy, though, WEP and CEP are the least efficient methods, as they require 1 and 2 additional jobs, respectively, in order to compute their pruning criteria.

For this strategy, the node-centric pruning schemes, CNP and WNP, are the most efficient ones, involving a single job. In contrast, they are the most time-consuming algorithms for the other strategies, since they process every edge twice, inside the neighborhoods of both adjacent nodes.

It is interesting to compare these patterns with the relative efficiency of pruning schemes in the case of serial processing. To this end, Figure 22 presents the Overhead Time of all serialized workflows over the DB_C dataset. The measurements were performed using the high-end server mentioned in Section 1 (Intel i7 3.40GHz, 64 GB of RAM, Debian Linux 7). Similar patterns were exhibited for the other datasets and are omitted for brevity.

First of all, we observe that the overhead of serial processing is significantly higher than that of parallel processing in the vast majority of cases. Second, the pruning schemes exhibit a similar behavior as in the case of the edge- and comparison-based strategies: the edge-centric ones, CEP and WEP, are significantly faster than their node-centric counterparts, CNP and WNP. However, the relations are different between cardinality- and weight-based schemes: CEP and CNP are faster than WEP and WNP, respectively, because the latter involve an additional iteration over the edges in order to estimate their pruning criterion. Thus, the overall most efficient serial algorithm is CEP, while WNP remains the most time-consuming one.

Weighting Schemes. For the edge-based strategy, CBS is the fastest weighting scheme, as the output value of its first `reduce` function in Stage 2 is empty. ARCS, ECBS, JS add information to this output value and, thus, require more time and I/O in order to process it. Given that they involve the same number of jobs, they exhibit similar overhead times. EJS requires two additional jobs in order to estimate the degree of every node, thus being the most time-consuming weighting scheme.

For the comparison-based strategy, we observe slightly different patterns. CBS, ECBS and JS yield similar overhead times, because they basically perform the same computation: for each pair of entities, they estimate the intersection of the associated block lists. They are faster than ARCS and EJS, as they rely exclusively on the information contained in the en-

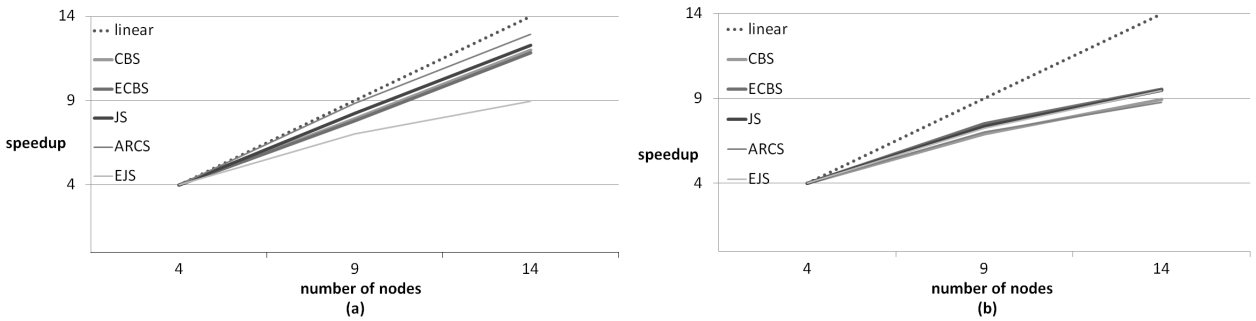


Figure 23: Speedup over DB_C of (a) the comparison-based strategy for WEP, and (b) the entity-based strategy for CNP.

riched input (i.e., the ids of the blocks associated with every entity). In contrast, EJS requires two additional jobs and is the most time-consuming weighting scheme in all cases. In most cases, ARCS lies between these two extremes, as it requires additional information and, thus, involves higher I/O than the most efficient schemes.

For the entity-based strategy, the differences between the weighting schemes are minor, except for EJS, which again requires an additional job and is, thus, the most time-consuming scheme. Among the other schemes, CBS and ARCS are slightly faster, since they do not load in memory the array with the number of blocks per entity, unlike JS and ECBS.

In the case of the serialized workflow, Figure 22 shows that ARCS is consistently the most time-consuming weighting scheme, because it produces very low values as edge weights (with tens of decimal digits). It is followed by EJS, which again involves higher computational cost in order to estimate the degree of every node. The remaining schemes share almost the same overhead, as their processing is very similar, computing the intersection of block lists.

11.4. Scalability

To assess the scalability of the comparison- and entity-based strategies, we estimate the *speedup* of their most efficient pruning schemes. That is, we measure the extent to which their overhead time decreases as we increase the number of available cluster nodes. Specifically, we apply the comparison-based WEP and the entity-based CNP to DB_C in combination with all weighting schemes. We increase the number of slave nodes from 4 to 9 and 14; in every case, there is an independent master node. The outcomes are presented in Figures 23(a) and (b) for WEP and CNP, respectively. In every figure, there is a dotted diagonal line, which illustrates the ideal case, where the speedup is linear to the number of nodes.

In Figure 23(a), we observe that all the weighting schemes show a speedup close to the ideal, with the exception of EJS. ARCS seems to be the weighting scheme that best exploits the available resources, showing a speedup of 12.92 when using 14 nodes. ECBS, CBS and JS have almost identical speedup values, ranging from 11.8 to 12.3, when using 14 nodes. This is because they basically perform the same computations, as explained previously. For EJS, the speedup is constantly lower than that of the other weighting schemes, because of the quadratic complexity of its additional jobs.

Regarding CNP, Figure 23(b) indicates that the deviation in the speedup of the various weighting schemes is much smaller than for WEP. Indeed, the speedup for 14 nodes fluctuates between 8.8 for ARCS and 9.5 for ECBS. This time EJS does not yield the worst speedup, as its additional job involves a linear complexity instead of a quadratic one.

In practice, these patterns indicate that the more cluster nodes we used, the faster was the execution of both strategies. For WEP, the improvement in time was almost as much as the the number of additional resources (until a certain point), while for CNP, n additional nodes improved the Overhead Time by $2n/3$ times. The reason is that WEP is able to balance the workload of its nodes right after Preprocessing, i.e., before applying Meta-blocking in Stage 3. In the case of CNP, though, this is impossible, since the workload of every `reduce` function in Stage 3 is not known a-priori.

11.5. Effectiveness

To assess the quality of the restructured blocks produced by Meta-blocking, we consider their performance with respect to the four relevant measures of Section 11.1. For every pruning scheme and dataset, we estimated the average value and the standard deviation of every measure across the five weighting schemes. The outcomes are presented in Figures 24(a) to (d). Remember that in all diagrams, the higher a bar is, the better is the corresponding performance. We should also note that all the MapReduce implementations are exact adaptations of their serialized counterparts, which means that the qualitative results of the serialized and the parallel implementations are identical.

Starting with Figure 24(a), we observe that the relative recall of the pruning schemes remains the same across all datasets: the node-centric ones, CNP and WNP, are more robust and 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. In fact, CEP and CNP they reduce the original *PC* by less than 10%, despite the significant enhancements in efficiency they convey.

Indeed, Figure 24(b) shows that WEP consistently 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,

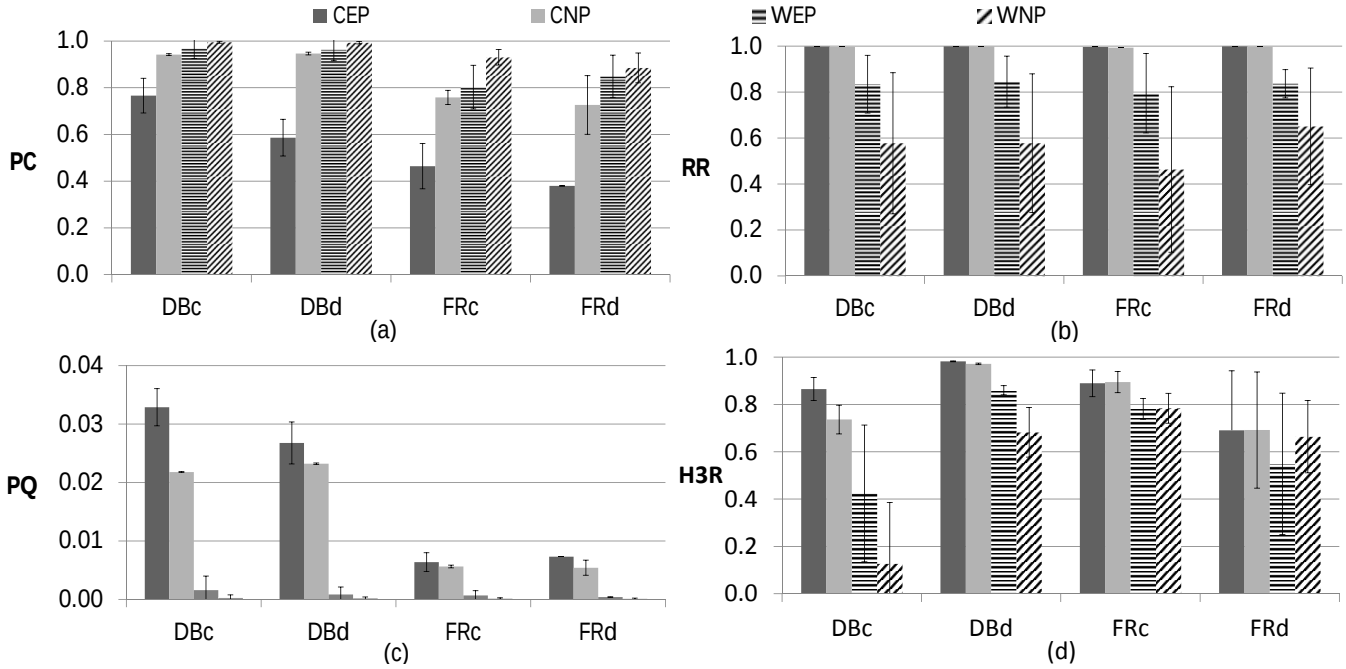


Figure 24: Average performance of the four pruning algorithms with respect to (a) *PC*, (b) *RR*, (c) *PQ*, and (d) *H3R*.

they perform such a deep pruning that they reduce the pairwise comparisons by 2 to 3 orders of magnitude across all datasets. This explains their poor recall.

Yet, Figure 24(c) demonstrates that CEP and CNP achieve significantly higher precision across all datasets. Compared to the input blocks, the restructured ones, produced by those schemes, increase precision by 2 to 3 orders of magnitude. For WEP and WNP, the improvement is slightly higher than an order of magnitude. This pattern actually indicates a clear trade-off between precision and recall: the higher *PQ* is for a specific method and dataset, the lower is the corresponding *PC* and vice versa.

To identify the scheme that achieves the overall best balance between the identified duplicates and the executed comparisons, we use *H3R*, which is presented in Figure 24(d). We observe that the cardinality-based methods, CEP and CNP, exhibit the highest values across all datasets, fluctuating between 0.97 and 0.67. The difference between the two methods is small, even though CNP retains twice as many comparisons as CEP, on average. Still, CNP should be preferred, since it retains the best comparisons per entity and, thus, is more robust to recall.

These patterns are in accordance with earlier findings about the relative performance of the four pruning schemes [4].

11.6. Discussion

The results of our experimental analysis demonstrate that the proposed strategies for parallel Meta-blocking yield significant improvements in the execution time, thus enabling ER in voluminous datasets. However, simple strategies cannot give us the full benefit: we observed that the edge-based strategy leads to significantly higher space requirements and is consistently slower than the comparison- and entity-based ones. The experiments also showed that our load balancing algorithm consistently outperforms the default balancer of Hadoop, assigning an almost identical workload to all the nodes of the cluster.

Among the four pruning schemes, the overall winner is CNP, as it involves the most efficient functionality (when using the entity-based strategy) and achieves the best balance between precision and recall in terms of *H3R* (CEP exhibits similar *H3R* values, but is significantly less robust to recall than CNP). The five weighting schemes exhibit similar effectiveness and are almost equivalent with respect to efficiency, with the exception of EJS, which is much slower and less scalable than the rest.

In summary, the edge-based strategy should not be used in practice. Instead, Parallel Meta-blocking should be applied using the comparison-based and entity-based strategies. The edge-centric algorithms, CEP and WEP, should always be combined with the comparison-based strategy, while the node-centric algorithms, CNP and WNP, should always be used with the entity-based strategy.

To demonstrate in a more intuitive way the actual benefit of Meta-blocking, we have estimated the times required to get the final matching results with and without Meta-blocking, given a blocking collection[14]. In the first case, we sum the times needed for the three stages of Meta-blocking and the time required to perform the resulting comparisons of Meta-blocking. In the latter case, we only estimate the time required to perform all the comparisons suggested by the input blocking collection. To estimate the time required for the comparisons, we performed 1 billion comparisons, using the Jaccard similarity of the tokens in the values of the DB_C collection. The average time required to get the similarity of 1 pair of entity profiles was $4.9 \cdot 10^{-7}$ minutes. Based on this number and taking as an example the CBS weighting scheme and the CNP pruning scheme, using the entity-based strategy, we esti-

[14]The efficiency of blocking has been already studied in [17, 19].

mate that the time required to perform Meta-blocking (including Block Filtering) and then the comparisons suggested by Meta-blocking is 76 minutes + $3.96 \cdot 10^7$ comparisons $\times$ $4.9 \cdot 10^{-7}$ minutes/comparison $\approx$ 95.5 minutes. The corresponding time required to perform the comparisons suggested by blocking, without using Meta-blocking, would be $4.23 \cdot 10^{10}$ comparisons $\times$ $4.9 \cdot 10^{-7}$ minutes/comparison = 20,727 minutes = 345.45 hours $\approx$ 14 days. The cost of using Meta-blocking, in this case, is a loss of 3.79% in *PC*.

12. Related Work

ER constitutes a well-studied problem that lies on the focus of numerous works. Comprehensive surveys of the relevant techniques can be found in [1, 20, 21, 22, 23, 24, 25]. Due to the inherently quadratic complexity of ER, a bulk of work aims for improving its scalability. We can distinguish two lines of research.

The first one includes *parallel ER methods*, which exploit the processing power of multiple cores in order to minimize the response time of ER. Early works towards this direction include [26, 27, 28]. More recent approaches are based on the MapReduce framework, which offers fault-tolerant, optimized execution for applications distributed across a set of independent nodes [8]. Based on this paradigm, [29, 30] introduce distributed approaches that improve the quality of entity matching through a recursive process: a decision about matching two entities triggers further decisions about matching their associated entities. Similar approaches are used by other iterative techniques, which employ some partial results of the ER process in order to locate new matches (e.g., [31, 32, 33, 34]).

A crucial aspect of MapReduce-based ER methods is the load balancing algorithm that distributes evenly the overall workload among the available nodes. Several recent works examine this aspect, with PairRange constituting the best solution so far [18].

Another approach is BlockSplit [18]. As its name suggests, it splits the bigger blocks into smaller sub-blocks and processes them in parallel, ensuring that every entity is compared to all entities in its sub-block, as well as to the entities of its superblock. BlockSplit has been proven to be less scalable and less generic than PairRange [18]: it needs to process multiple times the entity profiles of blocks that are split, creating an additional network and I/O overhead. Additionally, it may still lead to unbalanced workload, due to sub-blocks of different size.

A similar approach is followed by the dynamic blocking algorithm in [35]. Instead of perfectly balancing the load, though, its goal is to split large blocks into sub-blocks, "until they are all of tractable size". Yet, we already achieve this goal through Block Filtering, which completely removes large blocks (instead of splitting them into sub-blocks), as it considers them to be of lower importance.

Finally, two more load balancing algorithms were presented in [36]. Both rely on sketches in order to minimize memory consumption; the one aims to improve the space requirements of BlockSplit and the other of PairRange. In our case, though, all load balancing algorithms that were compared in Section 11.2 fit easily to the limited memory that is available to a single node. The reason is the optimized representation model, which represents every entity by an integer that denotes its id, while every block consists of a list of integers and is itself identified by a unique integer id.

The second line of research comprises *approximate techniques*, which focus on achieving a good balance between the number of identified duplicates and the number of executed comparisons. The most prominent among these approaches is *blocking*; it represents every entity by a set of blocking keys (i.e., signatures) and groups similar entities into blocks based on similar or identical keys. Comparisons are then executed only inside the resulting blocks. Depending on the definition of keys, blocking methods are distinguished into two categories:

(i) The *schema-based* ones target tabular data, i.e., data of high structuredness. They exploit a-priori schema knowledge in order to identify the attribute names with the most accurate and distinctive values and extract blocking keys of high quality from them. In this category fall methods like Suffix Arrays [37], StringMap [38], Standard Blocking [39], Canopy Clustering [40], Sorted Neighborhood [41] and Q-grams Blocking [42]. A comprehensive survey can be found in [3].

(ii) The *schema-independent* blocking methods target data of low structuredness, such as those stemming from the Web. They disregard schema knowledge when defining blocking keys, due to the unprecedented level of schema heterogeneity. For instance, Google Base[15] alone encompasses 100,000 distinct schemata that correspond to 10,000 entity types [43]. In this category fall methods like Token Blocking [6], Attribute Clustering [11], Total Description [15] and MFIBlocks [10]. They are capable of handling millions of entities, but still yield a large number of comparisons. Their scalability can be significantly enhanced by *Meta-blocking*, as explained above.

This work bridges the gap between the two lines of research for ER over Web Data, parallelizing Meta-blocking techniques to achieve even higher scalability. A similar effort for tabular data is made in [44, 45], which adapt Standard Blocking and Sorted Neighborhood, respectively, to MapReduce.

13. Conclusions

Despite the recent advances in blocking, ER remains a computationally intensive task with limited practical applications in the context of large data collections. In this paper, we parallelized Meta-blocking using MapReduce and enhanced dramatically the time efficiency of its serialized implementation. We proposed 3 parallelization strategies: *(i)* The edge-based one implements a straightforward approach that materializes the blocking graph; hence, it involves high I/O and high space requirements that do not scale well to large datasets. *(ii)* The comparison-based strategy offers a more elaborate implementation that uses the blocking graph implicitly. In this way, it reduces the overhead of data exchange and the number of required

[15] `http://www.google.com/base`

MapReduce jobs, leading to significant performance gains, especially for the edge-centric pruning schemes, CEP and WEP. *(iii)* The entity-based strategy is completely independent of the blocking graph, minimizing the data exchange and the overhead of MapReduce job chains. This approach offers an optimized implementation for the node-centric pruning schemes, CNP and WNP. All these strategies were combined with *MaxBlock*, a purpose-built load balancing algorithm that distributes the workload evenly among the cluster nodes. We also demonstrated their applicability in big data settings through an extensive experimental evaluation with the four largest, real datasets that have been applied to Meta-blocking.

In the future, we plan to adapt Supervised Meta-blocking [5] to MapReduce. Its functionality is fundamentally different from that of its unsupervised counterpart: first, it gathers a random sample of the blocking graph edges to use it as a training set and then, it applies a classification algorithm to the remaining edges. In this case, the main challenge is to parallelize the estimation of edge weights, which now comprise feature vectors with four dimensions.

Acknowledgements: This work was partially supported by the EU H2020 BigDataEurope (#644564), and EU FP7 SemData (#612551) projects.

References

[1] V. Christophides, V. Efthymiou, K. Stefanidis, Entity Resolution in the Web of Data, Synthesis Lectures on the Semantic Web: Theory and Technology, Morgan & Claypool Publishers, 2015.

[2] P. Christen, Data Matching - Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection, Data-Centric Systems and Applications, Springer, 2012.

[3] P. Christen, A survey of indexing techniques for scalable record linkage and deduplication, TKDE 24 (9) (2012) 1537–1555.

[4] G. Papadakis, G. Koutrika, T. Palpanas, W. Nejdl, Meta-blocking: Taking entity resolutionto the next level, IEEE Trans. Knowl. Data Eng. 26 (8) (2014) 1946–1960.

[5] G. Papadakis, G. Papastefanatos, G. Koutrika, Supervised meta-blocking, PVLDB 7 (14) (2014) 1929–1940.

[6] G. Papadakis, E. Ioannou, C. Niederée, P. Fankhauser, Efficient entity resolution for large heterogeneous information spaces, in: WSDM, 2011, pp. 535–544.

[7] G. Papadakis, G. Papastefanatos, T. Palpanas, M. Koubarakis, Scaling entity resolution to large, heterogeneous data with enhanced meta-blocking, in: EDBT, 2016.

[8] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.

[9] V. Efthymiou, G. Papadakis, G. Papastefanatos, K. Stefanidis, T. Palpanas, Parallel meta-blocking: Realizing scalable entity resolution over large, heterogeneous data, in: IEEE Big Data, 2015.

[10] B. Kenig, A. Gal, Mfiblocks: An effective blocking algorithm for entity resolution, Inf. Syst. 38 (6) (2013) 908–926.

[11] G. Papadakis, E. Ioannou, T. Palpanas, C. Niederée, W. Nejdl, A blocking framework for entity resolution in highly heterogeneous information spaces, IEEE Trans. Knowl. Data Eng. 25 (12) (2013) 2665–2682.

[12] L. Getoor, A. Machanavajjhala, Entity resolution: Theory, practice & open challenges, PVLDB 5 (12) (2012) 2018–2019.

[13] L. Getoor, A. Machanavajjhala, Entity resolution for big data, in: KDD, 2013.

[14] G. Papadakis, E. Ioannou, C. Niederée, T. Palpanas, W. Nejdl, Eliminating the redundancy in blocking-based entity resolution methods, in: JCDL, 2011, pp. 85–94.

[15] G. Papadakis, E. Ioannou, C. Niederée, T. Palpanas, W. Nejdl, Beyond 100 million entities: Large-scale blocking-based resolution for heterogeneous data, in: WSDM, 2012, pp. 53–62.

[16] G. Simonini, S. Bergamaschi, H. V. Jagadish, BLAST: a loosely schema-aware meta-blocking approach for entity resolution, PVLDB 9 (12) (2016) 1173–1184.

[17] V. Efthymiou, K. Stefanidis, V. Christophides, Big data entity resolution: From highly to somehow similar entity descriptions in the Web, in: IEEE Big Data, 2015.

[18] L. Kolb, A. Thor, E. Rahm, Load balancing for mapreduce-based entity resolution, in: ICDE, 2012, pp. 618–629.

[19] G. Papadakis, J. Svirsky, A. Gal, T. Palpanas, Comparative analysis of approximate blocking techniques for entity resolution, PVLDB 9 (9) (2016) 684–695.

[20] X. L. Dong, D. Srivastava, Big Data Integration, Synthesis Lectures on Data Management, Morgan & Claypool Publishers, 2015.

[21] A. Doan, A. Y. Halevy, Semantic integration research in the database community: A brief survey, AI Magazine 26 (1) (2005) 83–94.

[22] A. K. Elmagarmid, P. G. Ipeirotis, V. S. Verykios, Duplicate record detection: A survey, IEEE Trans. Knowl. Data Eng. 19 (1) (2007) 1–16.

[23] N. Koudas, S. Sarawagi, D. Srivastava, Record linkage: similarity measures and algorithms, in: SIGMOD, 2006.

[24] V. Efthymiou, K. Stefanidis, V. Christophides, Benchmarking blocking algorithms for web entities, IEEE Transactions on Big Data (to appear).

[25] E. Daskalaki, G. Flouris, I. Fundulaki, T. Saveta, Instance matching benchmarks in the era of linked data, Web Semantics: Science, Services and Agents on the World Wide Web 39 (2016) 1–14.

[26] P. Christen, T. Churches, M. Hegland, Febrl - A parallel open source data linkage system: http: //datamining.anu.edu.au/linkage.html, in: PAKDD, 2004, pp. 638–647.

[27] H. Kawai, H. Garcia-Molina, O. Benjelloun, D. Menestrina, E. Whang, H. Gong, P-swoosh: Parallel algorithm for generic entity resolution, Technical Report 2006-19, Stanford InfoLab (September 2006).

[28] H. Kim, D. Lee, Parallel linkage, in: CIKM, 2007, pp. 283–292.

[29] C. Böhm, G. de Melo, F. Naumann, G. Weikum, LINDA: distributed web-of-data-scale entity matching, in: CIKM, 2012, pp. 2104–2108.

[30] S. Lacoste-Julien, K. Palla, A. Davies, G. Kasneci, T. Graepel, Z. Ghahramani, Sigma: simple greedy matching for aligning large knowledge bases, in: KDD, 2013, pp. 572–580.

[31] O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, J. Widom, Swoosh: a generic approach to entity resolution, VLDB J. 18 (1) (2009) 255–276.

[32] I. Bhattacharya, L. Getoor, Collective entity resolution in relational data, TKDD 1 (1).

[33] X. Dong, A. Y. Halevy, J. Madhavan, Reference reconciliation in complex information spaces, in: SIGMOD, 2005.

[34] M. Herschel, F. Naumann, S. Szott, M. Taubert, Scalable iterative graph duplicate detection, IEEE Trans. Knowl. Data Eng. 24 (11) (2012) 2094–2108.

[35] N. McNeill, H. Kardes, A. Borthwick, Dynamic record blocking: efficient linking of massive databases in mapreduce, in: QDB, 2012.

[36] W. Yan, Y. Xue, B. Malin, Scalable load balancing for mapreduce-based record linkage, in: IPCCC, 2013, pp. 1–10.

[37] T. de Vries, H. Ke, S. Chawla, P. Christen, Robust record linkage blocking using suffix arrays, in: CIKM, 2009, pp. 1565–1568.

[38] L. Jin, C. Li, S. Mehrotra, Efficient record linkage in large data sets, in: DASFAA, 2003.

[39] I. P. Fellegi, A. B. Sunter, A theory for record linkage, Journal of the American Statistical Association 64 (328) (1969) 1183–1210.

[40] A. McCallum, K. Nigam, L. Ungar, Efficient clustering of high-dimensional data sets with application to reference matching, in: KDD, 2000, pp. 169–178.

[41] M. Hernández, S. Stolfo, The merge/purge problem for large databases, in: SIGMOD, 1995, pp. 127–138.

[42] L. Gravano, P. Ipeirotis, H. Jagadish, N. Koudas, S. Muthukrishnan, D. Srivastava, Approximate string joins in a database (almost) for free, in: VLDB, 2001, pp. 491–500.

[43] J. Madhavan, S. Cohen, X. L. Dong, A. Y. Halevy, S. R. Jeffery, D. Ko, C. Yu, Web-scale data integration: You can afford to pay as you go, in: CIDR, 2007, pp. 342–350.

[44] L. Kolb, A. Thor, E. Rahm, Dedoop: Efficient deduplication with hadoop, PVLDB 5 (12) (2012) 1878–1881.

[45] L. Kolb, A. Thor, E. Rahm, Multi-pass sorted neighborhood blocking with mapreduce, Computer Science - R&D 27 (1) (2012) 45–63.