Multi-core Meta-blocking for Big Linked Data

George Papadakis
University of Athens, Greece
gpapadis@di.uoa.gr

Konstantina Bereta
University of Athens, Greece
Konstantina.Bereta@di.uoa.gr

Themis Palpanas
Paris Descartes University, France
themis@mi.parisdescartes.fr

Manolis Koubarakis
University of Athens, Greece
koubarak@di.uoa.gr

ABSTRACT

Discovering matching entities in different Knowledge Bases constitutes a core task in the Linked Data paradigm. Due to its quadratic time complexity, Entity Resolution typically scales to large datasets through blocking, which restricts comparisons to similar entities. For Big Linked Data, Meta-blocking is also needed to restructure the blocks in a way that boosts precision, while maintaining high recall. Based on blocking and Meta-blocking, JedAI Toolkit implements an end-to-end ER workflow for both relational and RDF data. However, its bottleneck is the time-consuming procedure of Meta-blocking, which iterates over all comparisons in each block. To accelerate it, we present a suite of parallelization techniques that are suitable for multi-core processors. We present 2 categories of parallelization strategies, with each one comprising 4 different approaches that are orthogonal to Meta-blocking algorithms. We perform extensive experiments over a real dataset with 3.4 million entities and 13 billion comparisons, demonstrating that our methods can process it within few minutes, achieving high speedup.

ACM Reference format:
George Papadakis, Konstantina Bereta, Themis Palpanas, and Manolis Koubarakis. 2017. Multi-core Meta-blocking for Big Linked Data. In *Proceedings of Semantics2017, Amsterdam, Netherlands, September 11–14, 2017,* 8 pages.
https://doi.org/10.1145/3132218.3132230

1 INTRODUCTION

Entity Resolution (ER) constitutes a core task for Semantic Web, playing a major role in the realization of the fourth Linked Data principle [8]. Its goal is to identify and interlink (with `owl:sameAs` statements) all entity descriptions that pertain to the same real-world object, but are located in different Knowledge Bases (KBs). In this way, ER increases the value of Linked Open Data (LOD), enabling users and applications to use the resulting knowledge transparently. However, the LOD cloud involves KBs that are inadequately linked: less than 10% of them were strongly interlinked with at least another KB in 2014 [25]. One of the main causes is the high computational cost of ER, as all entities have to be compared to each other, yielding a quadratic time complexity.

Semantics2017, September 11–14, 2017, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5296-3/17/09…$15.00
https://doi.org/10.1145/3132218.3132230

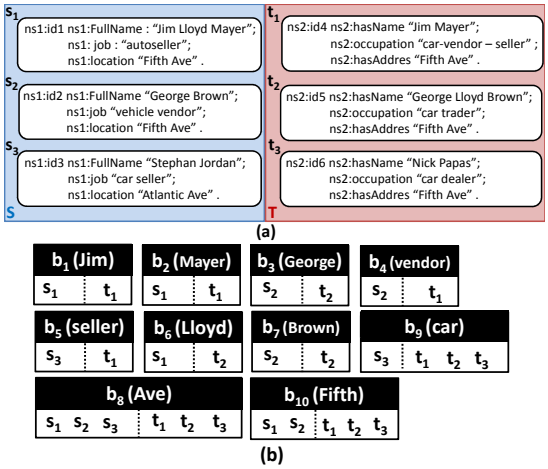

Figure 1: (a) Two KBs, S and T, with s_1 and s_2 matching with t_1 and t_2, resp. (b) The blocks produced by Token Blocking.

To enhance the efficiency of ER, blocking is typically used [2, 3]. In essence, blocking groups similar entities into blocks so that it suffices to execute comparisons only between those co-occurring in at least one block. In the context of LOD, blocking methods have to deal with extreme schema heterogeneity (i.e., *Variety*): there are ~2,600 diverse vocabularies, but only 109 of them are shared by more than one KB.[1] A simple, yet effective solution is to extract schema-free signatures from every entity and to create blocks based on their similarity or equality [19, 22].

As an example, consider the entities in Figure 1(a), where s_1 matches with t_1 and s_2 with t_2; Token Blocking [19] creates one block for every token that appears in the literal values of at least one entity in each KB, placing all relevant entities in it. The resulting blocks appear in Figure 1(b); both pairs of matching entities co-occur in at least one block, at the cost of 25 comparisons.

Such a high computational cost is common for blocking methods that use schema-free signatures. It is caused by two types of unnecessary comparisons [20]: the *redundant* ones repeatedly compare the same entities in different blocks, while the *superfluous* ones compare non-matching entities. In our example, the comparison between s_1 and t_1 in block b_2 belongs to the former category, first executed in b_1, and the comparison in b_5 to the latter one ($s_3 \neq t_1$).

Both types of unnecessary comparisons can be discarded with *Meta-blocking* [3, 5], the current state of the art blocking technique [22]. Based on the premise that the similarity of entities is reflected on the blocks they have in common, Meta-blocking restructures a

[1] http://stats.lod2.eu

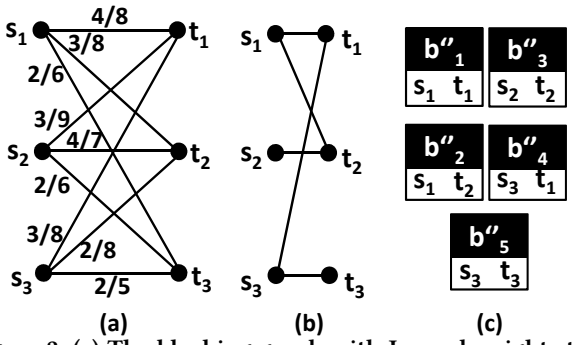

Figure 2: (a) The blocking graph with Jaccard weights that corresponds to the blocks in Figure 1(b), (b) the pruned blocking graph when using the average edge weight as threshold (0.35), (c) the corresponding restructured blocks.

set of blocks in order to reduce the number of comparisons by orders of magnitude at a small cost in recall (i.e., precision is significantly enhanced). First, it transforms the blocks into a graph that includes a node for every entity and an edge for every comparison. Then, it assigns a weight to every edge that is analogous to the number of blocks shared by the adjacent entities. Finally, it prunes the edges with low weights, creating a new block for every retained edge.

Continuing our example, Figure 2(a) depicts the graph corresponding to the blocks in Figure 1(b), with every edge weight expressing the Jaccard similarity of the set of blocks associated with its adjacent entities. Figure 2(b) illustrates the pruned blocking graph that results after discarding the edges with a weight lower than the average one. The restructured blocks appear in Figure 2(c); they encompass just 5 comparisons, with two of them involving the two pairs of matching entities, s_1-t_1 and s_2-t_2. Apparently, this is a major improvement over the original blocks in Figure 1(b) and the brute-force approach, which executes 9 pair-wise comparisons.

The combination of schema-free blocking methods and Meta-blocking lies at the core of JedAI Toolkit [23], which applies state-of-the-art ER methods developed by the database community to the RDF data of Semantic Web. JedAI can be used in three ways: *(i)* As an open-source library that combines these methods into an end-to-end ER workflow; *(ii)* As a user-friendly desktop application with a wizard-like interface that allows even lay users to build complex ER workflows with out-of-the-box solutions (i.e., without the need to fine-tune any configuration parameters); *(iii)* As a workbench for comparing the performance of various ER workflows over both structured (CSV, database) and semi-structured (RDF, XML) data.

At the moment, though, JedAI does not scale well to the *Volume* of *Big Linked Data*, i.e., to the large and increasing number and size of entity descriptions it involves; the LOD cloud alone contains almost 10,000 KBs with ~150B triples describing more than 55M entities[1]. In fact, Meta-blocking constitutes the bottleneck of JedAI's end-to-end ER workflow. The reason is that its computational cost is analogous to the number of comparisons in the blocks created by schema-free methods, which grow superlinearly with the number of entity descriptions, and the average number of blocks per entity, which grow linearly with the size of entity descriptions [6].

In this work, we enhance the time efficiency of Meta-blocking through a series of multi-core techniques that parallelize all its algorithms. Our multi-core techniques make the most of concurrent execution by involving a single state variable. This simplifies thread safety and results in a single point of synchronization, where every thread merely performs a quick, atomic operation. As a result, our methods exhibit high scalability and very low running times. Our extensive experiments demonstrate that they are capable of applying any Meta-blocking method to datasets with millions of entities and tens of billions of comparisons within a few minutes. In this way, JedAI is able to make the most of the commodity hardware that runs it, minimizing its response time without requiring high-end systems that are able to run MapReduce.

In summary, the contributions of this paper are the following:

- We propose 4 entity-based parallelization strategies, which rely on the optimized implementation of serialized Meta-blocking, offering low running times for applications with few available cores.
- We propose 4 block-based parallelization strategies, which rely on the original implementation of serialized Meta-blocking. They exhibit almost linear speedup, providing the fastest solution for applications with many cores.
- We perform exhaustive experiments that apply all parallelization strategies to all Meta-blocking algorithms on a dataset with 3 million entities. Our experiments identify the method with the lowest running time and the highest scalability. Our dataset along with the testing code (in Java), are publicly available.[2]

The rest of the paper is organized as follows: in Section 2, we discuss the main blocking and parallelization methods for ER, while in Section 3, we provide background knowledge on (Meta-)blocking. Section 4 introduces our parallelization methods, and Section 5 presents our experimental evaluation. In Section 6, we conclude the paper along with directions for future work.

2 RELATED WORK

Blocking methods can be distinguished in two categories. The first one includes *lossless methods*, which identify all pairs of entities that satisfy a set of link specifications. In this category typically fall schema-based methods that are integrated with Entity Matching, usually in the context of an ER framework. For example, Silk[3] includes MultiBlock [10], while LIMES[4] encompasses a series of blocking methods that rely on metric spaces, like HR[3] [16, 17]. These blocking methods typically deal with the Variety of LOD through user-defined link specifications, which lay the basis for designing blocking rules of high performance.

The second category involves *approximate blocking methods*, which sacrifice a small portion of the duplicate entities in order to execute a significantly lower number of comparisons. This category usually involves *stand-alone* blocking methods that focus on the literal values in entity descriptions and deal with the Variety of LOD through schema-free signatures [3, 19]. In this category fall practically all blocking methods that are inherently crafted for relational data [2]. A set of blocking methods for RDF data are discussed in [3], while [27] presents a more recent approach; initially, it performs unsupervised learning to identify the most discriminating properties and then, it extracts blocks from their literal values. Meta-blocking targets this type of blocking methods.

[2] http://sourceforge.net/projects/erframework
[3] http://silkframework.org
[4] http://aksw.org/Projects/LIMES.html

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**)	$EJS(e_i, e_j, B) = JS(e_i, e_j, B) \cdot \log\frac{\|V_B\|}{\|v_i\|} \cdot \log\frac{\|V_B\|}{\|v_j\|}$

Figure 3: Formal definition of Meta-blocking weighting schemes.

Another way of enhancing the efficiency of ER is parallelization. Early works towards this direction include [12, 13]. More recent approaches are based on the MapReduce framework. For example, [1] describes an iterative approach that employs partial results of ER in order to locate new matches. There is also a bulk of work on parallelizing blocking methods. For relational data, Standard Blocking and Sorted Neighborhood were adapted to MapReduce in [14] and [15], respectively. For RDF data, Silk MapReduce [11] and LIMESMR [9] perform ER over Hadoop, while LIMES is also able to exploit the massive parallelization capabilities of GPUs [18].

The work closest to ours is Parallel Meta-blocking [6, 7], which relies on the MapReduce paradigm and Apache Hadoop, in particular. The map phase creates the edges of the blocking graph or enriches the description of blocks, while the reduce phase performs edge weighting and pruning. Yet, Parallel Meta-blocking is orthogonal and complementary to our work, as MapReduce supports multi-core architectures. We also go beyond it in two ways:

(i) Parallel Meta-blocking relies on shared-nothing parallel processes that suffer from high disk overhead and high network I/O latencies. None of these factors affects our multi-core strategies, since they are designed for in-memory processing, employing multiple threads with shared memory.

(ii) Our multi-core parallelization strategies are easy to implement and adjust and have minimum hardware requirements, running on any multi-core processor. In contrast, Parallel Meta-blocking requires significant effort and resources to setup a large infrastructure that exploits MapReduce. It also needs ample time to optimize Hadoop, given that there are at least 250 tunable parameters that affect the performance of a Hadoop cluster [4].

3 PRELIMINARIES

An entity with URI `i` is symbolized by e_i and its description comprises all triples of the form `<i p o>`. Entity Resolution can be defined as the task of matching all entities from a source KB S with all entities from the target KB T. Apparently, the computational cost of this procedure is $|S| \times |T|$ comparisons.

Blocking restricts this computational cost by clustering similar entities into blocks and performing comparisons only between the co-occurring entities. A set of blocks B is called *block collection* and $|B|$ stands for its size. A block with index k is represented by b_k and internally consists of two inner blocks: $b_{k,s} \in S$ and $b_{k,t} \in T$. Both inner blocks should be non-empty. The block size indicates the number of contained entities, $|b_k|=|b_{k,s}|+|b_{k,t}|$, while the block cardinality indicates the number of contained comparisons, $||b_k||=|b_{k,s}|\times|b_{k,t}|$. Similarly, we define the cardinality of B as: $||B||=\sum_{b_j \in B} ||b_j||$. $B_i \subseteq B$ denotes the blocks involving entity e_i, with $|B_i|$ indicating its size. An individual comparison between entities e_i and e_j is symbolized by $c_{i,j}$.

Meta-blocking is a suite of algorithms that exclusively apply to *redundancy-positive* block collections, where the similarity of two entity profiles is proportional to the number of blocks they have in common [5, 20, 21]. That is, the more blocks two entities share, the more likely they are to be matching.

In more detail, Meta-blocking restructures a redundancy-positive block collection B into a new one with higher precision and equivalent recall. To this end, it operates on the level of individual comparisons. Central to this procedure is an undirected bipartite graph, called *blocking graph*, where the nodes correspond to entities and the edges connect those co-occurring in blocks. The blocking graph is simple, involving no parallel edges between the co-occurring entities; thus, it eliminates all *redundant* comparisons.

To discard part of the *superfluous* comparisons, the edges of the blocking graph are weighted so as to reflect the similarity of the blocks shared by adjacent entities. Five generic, schema-agnostic weighting schemes were proposed in [20]: ARCS, CBS, ECBS, JS, EJS. Their formal definitions appear in Figure 3(a), where V_B denotes the total number of nodes in the blocking graph of B and v_i the node degree corresponding to entity e_i. In all cases, higher weights indicate adjacent entities that are more likely to be matching, with low-weighted edges signaling probably superfluous comparisons.

There are 4 main algorithms for pruning edges with low weights:

- *Weight Edge Pruning* (WEP) iterates over all edges and discards those with a weight lower than a *global* threshold, which is equal to the average edge weight of the entire blocking graph.
- *Weight Node Pruning* (WNP) iterates over all nodes and discards the adjacent edges with a weight lower than a *local* threshold, which is equal to the average edge weight in each node neighborhood.
- *Cardinality Edge Pruning* (CEP) retains the top K weighted edges of the entire graph, where $K = \sum_{b_i \in B} |b_i|/2$.
- *Cardinality Node Pruning* (CNP) retains the top k weighted edges in the neighborhood of each node, with $k = \sum_{b_i \in B} \frac{|b_i|}{(|S|+|T|)}$.

Theoretically, the node-centric pruning algorithms, WNP and CNP, might produce restructured blocks that still contain redundant comparisons: the same edge might be kept in the neighborhoods of both adjacent entities. In this work, we exclusively consider the implementation of *Redefined Node-centric Pruning* [21], which does not retain such redundant comparisons.

Another approach is *Reciprocal Node-centric Pruning* [21], which treats the redundant retained edges as strong indications for matches: if a pair of entities is reciprocally connected in the pruned blocking graph that is produced by WNP or CNP, they are highly likely to be matching; thus, Reciprocal WNP and CNP retain one comparison only for such entity pairs. We do not consider these two reciprocal pruning methods, since the parallelization strategies we propose for WNP and CNP apply equally to them, without any modification.

On the whole, Meta-blocking boosts the precision of redundancy-positive block collections at a limited cost in their recall [5, 20, 21]. It comprises 4 main pruning algorithms, with CEP and CNP accommodating applications that aim minimize running time and maximize precision (e.g., Pay-As-You-Go ER), while WEP and WNP accommodate applications emphasizing recall at the cost of higher running times [21]. Every pruning algorithm can be combined with 5 weighting schemes, thus yielding 20 pruning schemes, in total. We propose parallelization approaches for all of them.

4 APPROACH

We now present a set of multi-core techniques for parallelizing the main pruning algorithms of Meta-blocking. In their description, we use the terms comparison and blocking graph edge interchangeably.

4.1 Parallelization Strategies

At the core of Meta-blocking lies edge weighting and the estimation of the number of blocks shared by a pair of neighboring entities, in particular. Based on the two implementations for this process [21], we define two categories of parallelization strategies.

The first category involves **block-based methods**. They iterate over the input blocks B and for each comparison $c_{i,j}$ in $b_k \in B$, they compute the intersection of the blocks associated with the entities e_i and e_j using the *Entity Index* (this data structure is an inverted index that associates every entity id with the ids of the blocks that contain it). Methods of this category parallelize the Original Edge Weighting approach [21].

The second category comprises **entity-based methods**. They iterate over the input KBs and for each entity $e_i \in (S \cup T)$, they aggregate its neighbors from all associated blocks, B_i. The number of occurrences of each neighbor is recorded, as it is equal to the number of common blocks. This category parallelizes the Optimized Edge Weighting approach [21].

For both categories, we consider 4 parallelization strategies that essentially rely on the same principle: the computational cost is split into a set of *chunks* that are placed in an *array*, with an *index* indicating the next chunk to be processed. Every thread retrieves the current value of the index and is assigned to process the corresponding chunk. The index is then incremented until it reaches the end of the chunk array. At that point, it returns a negative value to every request, thus terminating the respective thread.

This approach has two benefits: *(i)* There is a single state variable, the index of the chunk to be processed, which simplifies thread safety. All chunks can be read from the array by any thread without affecting the state of the others, as their content is not altered in any way. *(ii)* There is a single atomic synchronized operation, the retrieval of the value of the index. This enhances concurrency, since each thread holds the lock for rather short time, performing the costly operation (chunk processing) outside the synchronized block.

In this context, we propose the following four parallelization strategies. Note that for convenience, we use the term *item collection* (I) to refer either to a block collection or a set of entities. An individual *item* with id k is denoted by i_k, with $||i_k||$ symbolizing the number of comparisons it involves; if the item corresponds to a block b_l, $||i_k||$ indicates the number of comparisons it involves ($||i_k||$=$||b_l||$), while for an entity e_m, $||i_k||$ stands for the total number of comparisons involving e_m, i.e., $||i_k||$=$\sum_{b_n \in B_m} |b_{n,t}|$ if $e_m \in S$ and $||i_k||$=$\sum_{b_n \in B_m} |b_{n,s}|$ if $e_m \in T$. Note also our parallelization strategies differ in two more respects (in addition to the way they count common blocks): the definition of the chunks, and the creation of the chunk array.

(i) **Random parallelization**. Every chunk corresponds to an individual item and its position in the array is arbitrarily determined. Consequently, the computational cost of the individual chunks differs widely. This may result in consecutive chunks of very low cost (few comparisons), causing the threads to frequently wait for the lock, before retrieving the current value of the index. Another problem is that the largest chunk may be placed in one of the last positions in the array, thus causing significant waiting time.

Algorithm 1: Clustering Items Into Partitions.

Input: I the input item collection
Output: P the set of partitions

```
1  I' ← sort(I); // sort items in decreasing cardinality
2  i_0 ← I'.remove(0); // remove largest item
3  maxCost ← ||i_0||; // max comparisons per partition
4  P_0 ← {i_0}; // first partition
5  Q ← {P_0}; // priority queue, sorting partitions in increasing cost
6  while I' ≠ {} do // while not empty
7      i_0 ← I'.remove(0); // remove current first item
8      P_head ← Q.poll(); // get lowest cost partition
9      totalCost ← ||i_0|| + P_head.currentCost();
10     if totalCost ≤ maxCost then
11         P_head ← P_head ∪ {i_0}; // add to partition
12     else
13         P_i ← {i_0}; // create new partition
14         Q.add(P_i); // add to queue
15     Q.add(P_head); // place back to queue
16 return Q.getElements();
```

(ii) **Naive parallelization**. The chunks correspond to individual items, but they are placed in the array in decreasing order of cost: the first position is occupied by the item with the most comparisons, while the last place contains the item with the least comparisons; in case of tie, the order is determined arbitrarily. The distribution of computational costs is heavily skewed towards few comparisons for both blocks and entities, as shown in Tables 3(a) and (b), respectively. Hence, the cost assigned to every thread differs substantially at the beginning of the chunk array, which involves a large part of the overall comparisons. Significant waiting for the lock is expected to take place at the end of the chunk array, but there most items entail just a single comparison.

(iii) **Partition parallelization**. The idea behind this approach is to reduce the waiting for the lock by clustering items into larger chunks that have similar computational costs. We call these chunks *partitions* and we derive them from Algorithm 1. This approach exploits the heavily skewed distribution of comparisons in blocks and entities: it sorts all items in decreasing cost (Line 1) and sets the maximum computational cost of every partition equal to the number of comparisons in the first item (Lines 2-3). It creates a partition for this item and places it in the priority queue Q, which sorts all partitions in ascending cost (Lines 4-5); this means that its head always corresponds to the partition with the least comparisons. Then, our algorithm removes iteratively the first item still in the sorted collection I, i_0, and examines whether it fits in the head partition P_{head} (Lines 6-10). If it does, i_0 is added to P_{head} (Line 11); otherwise, if the combined costs of i_0 and P_{head} exceed the maximum computational cost per partition, a new partition is created, containing only i_0 (Lines 12-13). Both partitions are then placed back in the queue (Lines 14-15). At the end, the resulting partitions are placed in the chunk array in decreasing order of cost so that the processing starts from the largest ones.

The functionality of Algorithm 1 is illustrated in Figure 4(b). It scales well to large item collections, as its time complexity is dominated by the sorting in Line 1, i.e., $O(|I| \cdot log|I|)$ – the overall cost for inserting one partition in Q for every item is $O(|I| \cdot log|Q|)$, where $|Q|(< |I|)$ stands for the size of the priority queue (i.e., the number of partitions). Note that Algorithm 1 is inspired from the Load

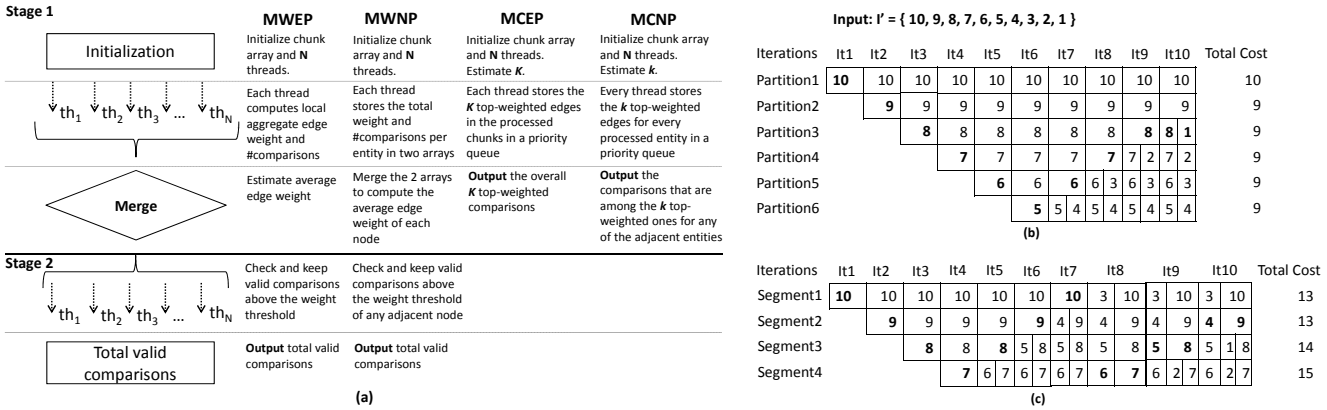

Figure 4: (a) Execution plan for all parallelization strategies and pruning algorithms. (b) Applying Algorithm 1 to a set of items I', ordered in decreasing computational cost. (c) Applying Algorithm 2 to the same set of items. In (b) and (c), the head of the priority queue in every iteration is marked with bold and italics.

Algorithm 2: Clustering Items Into Segments.

Input: I the input item collection, N the number of cores
Output: S the set of segments

```
1 Q ← {}; // priority queue, sorting segments in increasing cost
2 for i ← 1 to N do
3     Q.add(S_i); // add an empty segment per core
4 while I' ≠ {} do // while not empty
5     i_0 ← I'.remove(0); // remove first item
6     S_head ← Q.poll(); // get segment with lowest cost
7     S_head ← S_head ∪ {i_0}; // add to segment
8     Q.add(S_head); // place back to queue
9 return Q.getElements();
```

Balancing algorithm of [6], but there it is used only for clustering blocks into partitions, whereas we apply it to entities, too.

(iv) **Segment parallelization**. The motivation behind this approach is to develop a thread confinement (i.e., shared-nothing) parallelization strategy that eliminates the waiting time for the lock [24]. This can be accomplished by grouping the input items into as many clusters as the number of available threads such that all clusters have the same computational cost. Every cluster is called *segment* and its cost is approximately $||I||/N$, where N is the number of threads and $||I||$ stands for the total number of comparisons in I (i.e., $||I|| = \sum_{i_k \in I} ||i_k||$).

To split the input items into segments, we apply Algorithm 2. Initially, it creates a priority queue with one empty segment per thread (Lines 1-3). The queue sorts the segments in increasing cost, from the least comparisons to the most ones. Then, our algorithm iterates over the input items and places each of them in the segment located at the head of the queue (Lines 4-8). The resulting segments are then placed in the chunk array and each thread undertakes one of them by retrieving the value of the index just once, at the beginning of its processing. This functionality is illustrated in Figure 4(c). Note that the input item collection is already ordered in decreasing cost, but this is not required. Note also that the time complexity of Algorithm 2 is $O(|I|)$, as it simply iterates over the input items (the overall cost for inserting a segment for each item into the priority queue Q is $O(|I| \cdot logN) \approx O(|I|)$, where N is the number of cores).

In total, we propose 8 parallelization strategies for each pruning algorithm and weighting scheme. The relative performance of every strategy cannot be determined a-priori, as it depends on the characteristics of the input data. The reason is that the computational cost of a chunk rarely coincides with the number of comparisons that are associated with the corresponding entity or block: some comparisons are skipped, because they are redundant, while others are quickly pruned, due to their low weights. The higher the portion of unnecessary comparisons is, the less accurate is the a-priori estimation of the cost of individual chunks and the less balanced is the workload assigned to each thread in every iteration.

4.2 Multi-core Execution Plan

We now introduce the general parallelization strategy for each pruning algorithm of Meta-blocking, which applies uniformly to all parallelization strategies presented above. The execution plan of this strategy is illustrated in Figure 4(a). It distinguishes the parallel algorithms into **single-stage** and **two-stage** ones, according to the number of phases they involve.

To the latter category belongs *Multi-core Weighted Edge Pruning* (MWEP), as it iterates over the edges of the blocking graph twice: once for estimating the average weight and once for pruning the edges that do not exceed it. Similarly, *Multi-core Weighted Node Pruning* (MWNP) traverses the nodes of the blocking graph twice: the first iteration estimates the average edge weight in every node neighborhood, while the second one prunes all edges below the threshold in both adjacent node neighborhoods.

In contrast, *Multi-core Cardinality Edge Pruning* (MCEP) is a single-stage algorithm: it iterates over the edges of the blocking graph just once in order to fill a priority queue with the top-K weighted edges. *Multi-core Cardinality Node Pruning* (MCNP) also needs a single iteration over the nodes of the graph to fill a priority queue with the top-k weighted edges for each entity. Both MCEP and MCNP determine their cardinality threshold with a quick iteration over the input blocks that does not require parallelization.

In this context, the multi-core execution plan comprises 5 steps:

(i) **Initialization**. This step is common for all algorithms, building the chunk array, initializing the index to the first place and setting off N threads.

(ii) **First parallel phase**. In this step, each thread executes iteratively a parallel task that processes the next available chunk; it

traverses the comparisons in the chunk and estimates the weight for every valid (i.e., non-redundant) comparison. Then, it updates accordingly a set of variables or data structures. • In MWEP, every thread conveys two local counters: the number of executed comparisons and the aggregate weight of the executed comparisons. • In MWNP, each thread contains two local arrays of size $|S|+|T|$, i.e., they have one cell for every input entity. This is necessary for both block- and entity-based parallelization strategies: in the former case, a single thread may process comparisons that involve all input entities, whereas in the latter case, we do not know a-priori which entities will be assigned to every thread. The first array accumulates the number of comparisons involving the corresponding entity, while the second one accumulates the respective edge weights. • In MCEP, each thread maintains a local priority queue that sorts comparisons in increasing weight (i.e., the head element corresponds to the edge with the lowest weight) and contains up to K elements. Every valid comparison is added in the priority queue and if the overall size of the queue exceeds K, the head element is removed. • The same procedure applies to the threads of MCNP, with the only difference being that every thread maintains a priority queue with k elements for each input entity ($k \ll K$).

(iii) **Merge**. This phase aggregates the outcomes of the N threads after all of them conclude their processing. • MCEP merges the N priority queues into a single one with the globally top-K weighted comparisons and returns it as output. • MCNP merges the N priority queues of every entity to estimate the k adjacent edges with the highest weights; then, it returns as output all comparisons that are top-weighted for any of the entities they involve. • In MWEP, the average edge weight is estimated by dividing the sum of aggregate weights by the sum of executed comparisons. • The same happens in MWNP, which creates a global array with the average edge weight in the neighborhood of every entity. For the last two algorithms, the index is subsequently reset to the first element of the chunk array and the N threads are restarted.

(iv) **Second parallel phase**. In this phase, MWEP refines the input items by retaining locally only the valid comparisons that exceed the average edge weight. In MWNP, a valid comparison is retained if it surpasses the weight threshold of any adjacent node.

(v) **Second merge**. The final step of the two-stage algorithms assembles the comparisons retained locally in every thread and returns them as output.

5 EXPERIMENTAL EVALUATION

We now present our experimental evaluation, which applies all multi-core Meta-blocking algorithms described above in combination with all parallelization strategies to a large, real LOD dataset. Its goal is twofold: *(i)* to identify the most efficient parallelization strategy in terms of absolute execution time, and *(ii)* to identify the strategy that scales better as the number of physical cores increases. Note that evaluating the performance of Meta-blocking with respect to effectiveness lies out of our scope, as this has already been examined in a series of previous works, which employed numerous established datasets [20–22, 26].

Experimental Setup. All methods and experiments were implemented in Java, version 8. All experiments were performed on a server with Ubuntu 12.04, 32GB RAM and 2 Intel Xeon E5620 processors, each having 4 physical cores and 8 logical cores at 2.40GHz.

	DBPedia3.0rc	DBPedia3.4
Entities	1,190,733	2,164,040
Duplicates	892,579	
Triples	$1.69 \cdot 10^7$	$3.50 \cdot 10^7$
Predicates	30,757	52,554
$\|S\| \times \|T\|$	$2.58 \cdot 10^{12}$	

(a)

	Input Blocks
Blocks	1,239,066
Matches	890,817
Recall	0.998
Precision	$6.86 \cdot 10^{-5}$
$\|\|B\|\|$	$1.30 \cdot 10^{10}$

(b)

Table 1: Technical characteristics of (a) the DBPedia versions that are resolved, and (b) the blocks that are given as input to Meta-blocking.

For every parallelization strategy, we used 2, 4, 6 and 8 cores. Yet, we cannot be sure that every thread is assigned to a physical core, as we did not intervene in the functionality of the operating system. Therefore, our time measurements offer a pessimistic estimation of the actual performance of our algorithms. To reduce the effect of external factors (e.g., disk workload), we repeated every measurement 3 times and consider the resulting average performance.

Measures. Our evaluation criteria are the following:

(i) Wall-clock running time (WCT). This measure estimates the overhead time of Meta-blocking, i.e., the time that intervenes between receiving a block collection as input and returning the restructured blocks as output. Lower values correspond to higher efficiency for Meta-blocking. Note that this measure applies both to serialized and parallel Meta-blocking algorithms.

(ii) Speedup. This measure estimates the scalability of Multi-core Meta-blocking, expressing the extent to which its wall-clock running time decreases as we increase the number of available cores. Assuming a set of core numbers $\{n_{min}, \ldots, n_{max}\}$, speedup takes values in the interval $[n_{min}, n_{max}]$ and is formally defined it as: $speedup(n_i) = n_{min} \times WCT(n_{min})/WCT(n_i)$, where $WCT(x)$ is the wall-clock running time of multi-core Meta-blocking on top of x cores. Apparently, the closer speedup is to n_{max}, the better is the scalability of multi-core Meta-blocking, with $speedup(n_{max}) = n_{max}$ corresponding to the ideal case.

Dataset. We perform our experiments on top of the large KB of DBPedia[5], which lies at the center of the LOD cloud [25]. We actually resolve two snapshots of the English DBPedia Infoboxes that chronologically differ by 2 years, versions 3.0rc and 3.4. Table 1(a) presents their technical characteristics. In total, they involve almost 3.4 million entities that are described by 52 million triples and 55,000 distinct predicates. Among them, less than a million are duplicates, having the same URI in both snapshots. Note that we use this dataset only to evaluate time efficiency, illustrating the performance of our parallelization strategies, since the same dataset has been repeatedly used for assessing effectiveness [20–22, 26].

To extract blocks from this dataset, we apply Token Blocking and refine its blocks with Block Purging and Block Filtering. The former discards the blocks that contain more than half the input entities, while the latter retains every entity in 80% of its smaller blocks. Both methods are commonly applied to schema-free blocking methods in order to reduce the total number of block comparisons, $||B||$, to manageable levels for serialized processing, without any significant impact on recall [20–22, 26]. The technical characteristics of the resulting blocks appear in Table 1(b). They achieve almost perfect recall, while executing two orders of magnitude less comparisons than the brute-force approach ($|S| \times |T|$ in Table 1). However, their

[5] http://wiki.dbpedia.org

precision remains very low: almost 15,000 comparisons have to be executed in order to identify a new match.

Qualitative Performance. Due to space restrictions, in the following we exclusively combine Meta-blocking algorithms with the top performing weighting scheme, namely ARCS [20, 21]. The other weighting schemes exhibit similar behavior with respect to the relative performance of our parallelization strategy.

For the sake of completeness, we present in Table 2 the effectiveness of the four main pruning algorithms in combination with ARCS. We observe that CEP retains approximately 6 comparisons per entity for a recall around 79%, while CNP executes less than 10 comparisons per entity for a recall higher than 96%. In this way, both algorithms raise precision by 3 orders of magnitude. WEP and WNP maintain the original recall, while saving 1 to 2 orders of magnitude more comparisons than the brute-force approach; hence, they raise precision to a similar extent.

Tables 3(a) and (b) present the distributions of computational cost before and after applying Algorithms 1 and 2 on the input blocks and entities, respectively. Each distribution is described by the minimum, the median and the maximum comparisons per item or chunk. The first column in each table sketches the original distributions of comparisons, which are heavily skewed towards small costs: most items involve very few comparisons, with their frequency decreasing as we move to larger comparisons. These distributions are flattened by Alg. 1, which creates partitions with a cost identical with the maximum comparisons per block or entity. In total, it creates 458 and 15,333 partitions for the input blocks and entities, respectively. In contrast, Alg. 2 distributes the computational cost evenly among all segments it creates, regardless of the number of available cores. Note that for each number of cores, its segments have the same computational cost for blocks and entities.

Time Performance. To assess the time efficiency of our techniques, we estimated *WCT* and speedup for all parallelization strategies in combination ARCS and the 4 multi-core Meta-blocking algorithms. The results are depicted in Figure 5. We also report *WCT* for the serialized pruning algorithms in Table 2, while Table 3(a) and (b) presents *WCT* for Algorithms 1 and 2 over blocks and entities, respectively. We observe the following patterns:

(i) Following the relative performance of Original and Optimized Edge Weighting, the entity-based strategies are significantly faster than the corresponding block-based ones with respect to *WCT*. This pattern is consistent across all pruning algorithms.

(ii) The block-based strategies consistently achieve a significantly higher speedup than the entity-based ones. This means that they scale much better, having the potential to outperform entity-based strategies with respect to *WCT*, too, for large numbers of cores. The reason is that the number of comparisons provides a more reliable estimation for the computational cost of blocks, than for entities. For example, it is possible for an entity e_i to co-occur with a single entity e_j in 5 blocks, in which case only 1 out of 5 comparisons involving these two entities is valid and a single edge weight has to be computed; for an individual block, though, it is very unlikely that 80% of its comparisons are redundant. As a result, the workload of blocks is more evenly balanced among the available nodes than the workload of entities.

	CEP	CNP	WEP	WNP
$\|\|B'\|\|(\times 10^7)$	2.11	3.30	14.27	72.21
Recall	0.788	0.963	0.901	0.992
Precision$(\times 10^{-3})$	33.41	26.06	5.64	1.23
$WCT(\times 10^6$ msec)	4.39	12.48	10.96	19.92

Table 2: Performance of the main pruning algorithms in combination with the ARCS weighting scheme.

	Original I	Partitions	2-Seg.	4-Seg.	6-Seg.	8-Seg.
Min.	1	$2.1{\cdot}10^7$	$6.5{\cdot}10^9$	$3.3{\cdot}10^9$	$2.2{\cdot}10^9$	$1.6{\cdot}10^9$
Median	2	$2.8{\cdot}10^7$	-	$3.3{\cdot}10^9$	$2.2{\cdot}10^9$	$1.6{\cdot}10^9$
Max.	$2.8{\cdot}10^7$	$2.8{\cdot}10^7$	$6.5{\cdot}10^9$	$3.3{\cdot}10^9$	$2.2{\cdot}10^9$	$1.6{\cdot}10^9$
WCT	-	306	66	81	89	106

(a) Blocks

	Original I	Partitions	2-Seg.	4-Seg.	6-Seg.	8-Seg.
Min.	1	$3.0{\cdot}10^5$	$6.5{\cdot}10^9$	$3.3{\cdot}10^9$	$2.2{\cdot}10^9$	$1.6{\cdot}10^9$
Median	7,019	$8.5{\cdot}10^5$	-	$3.3{\cdot}10^9$	$2.2{\cdot}10^9$	$1.6{\cdot}10^9$
Max.	$8.5{\cdot}10^5$	$8.5{\cdot}10^5$	$6.5{\cdot}10^9$	$3.3{\cdot}10^9$	$2.2{\cdot}10^9$	$1.6{\cdot}10^9$
WCT	-	238,064	84	87	99	106

(b) Entities

Table 3: Distribution of comparisons per (a) block and (b) entity before and after applying Algorithms 1 and 2. *WCT* is measured in milliseconds (msec).

(iii) Table 3 shows that Algorithm 2 is much faster than Algorithm 1 over both blocks and entities, due to its lower time complexity ($O(|I|)$ vs. $O(|I|\log|I|)$. Algorithm 1 takes significantly more time for entities than for blocks, as in the former case it produces 30 times more partitions. In contrast, Algorithm 2 is slightly faster over blocks than over entities, due to the lower number of items (1.2 million blocks vs. 3.3 million entities, cf. Table 1). Nevertheless, the running time of both algorithms accounts for a negligible portion of the overall *WCT* for most parallelization schemes, which consistently exceeds 10^6 milliseconds, regardless of the available cores (cf. the diagrams in the upper row of Figure 5).

(iv) CEP exhibits the lowest speedup across all pruning algorithms, especially for entity-based strategies. The reason is that the non-parallelizable cost of merging the N priority queues increases linearly with N, i.e., the number of cores. Especially for entity-based methods, this cost rises to 20% of *WCT* for 8 cores.

(v) Compared to Optimized Edge Weighting, the fastest serialized implementation of Meta-blocking, all parallelization strategies achieve significantly lower running times than those reported in Table 2, already for 2 cores.

(vi) Among the eight strategies, entity-based segment parallelization consistently exhibits the best performance with respect to *WCT*: it requires less than 30 minutes ($1.8{\cdot}10^6$ milliseconds) to apply any Meta-blocking algorithm, when using 8 cores. Its advantages are the very low non-parallelizable cost (Algorithm 2) and its shared-nothing functionality, which avoids waiting for the lock.

6 CONCLUSIONS

In this work, we presented 8 parallelization strategies for performing Meta-blocking in multi-core settings. Our methods are distinguished into block- and entity-based ones, but they all share the same core functionality: there is a single state variable that is incremented through a single, very fast point of synchronization. Thus, they achieve high concurrency, reducing the running time proportionally to the number of available cores. In absolute terms,

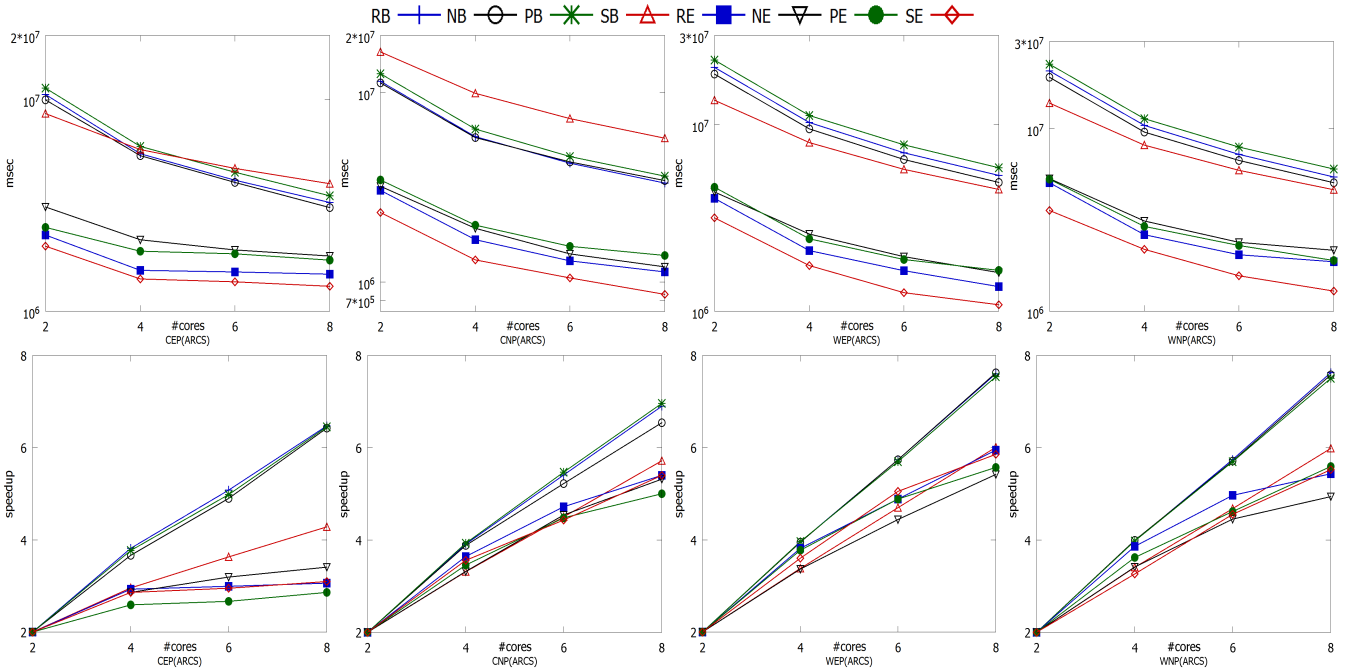

Figure 5: *WCT* (on the first row) and speedup (on the second row) for all multi-core strategies of the four main Meta-blocking algorithms in combination with the ARCS weighting scheme. *RB* and *RE* stand for the random parallelization strategy on blocks and entities, resp., while *Nx*, *Px* and *Sx* denote the naive, the partition and the segment parallelization strategies, respectively.

entity-based segment parallelization is the fastest approach across most pruning algorithms and weighting schemes, requiring less than 30 minutes to process a large dataset with 3.4 million entities/nodes and 13 billion comparisons/edges. Hence, it offers the best solution for ER applications with limited resources, like JedAI.

In the future, we plan to integrate Multi-core Meta-blocking with Parallel Meta-blocking on the basis of advanced MapReduce frameworks, like Apache Spark and Flink.

Acknowledgements. This work has been partially supported by the project "Copernicus App Lab", which is funded by the EU Horizon 2020 programme under grant agreement No. 730124.

REFERENCES

[1] Christoph Böhm, Gerard de Melo, Felix Naumann, and Gerhard Weikum. 2012. LINDA: distributed web-of-data-scale entity matching. In *CIKM*. 2104–2108.
[2] Peter Christen. 2012. A Survey of Indexing Techniques for Scalable Record Linkage and Deduplication. *TKDE* 24, 9 (2012), 1537–1555.
[3] Vassilis Christophides, Vasilis Efthymiou, and Kostas Stefanidis. 2015. *Entity Resolution in the Web of Data*. Morgan & Claypool Publishers.
[4] Xu Chu, Ihab F. Ilyas, and Paraschos Koutris. 2016. Distributed Data Deduplication. *PVLDB* 9, 11 (2016), 864–875.
[5] Xin Luna Dong and Divesh Srivastava. 2015. *Big Data Integration*. Morgan & Claypool Publishers.
[6] Vasilis Efthymiou, George Papadakis, George Papastefanatos, Kostas Stefanidis, and Themis Palpanas. 2015. Parallel meta-blocking: Realizing scalable entity resolution over large, heterogeneous data. In *IEEE Big Data 2015*. 411–420.
[7] Vasilis Efthymiou, George Papadakis, George Papastefanatos, Kostas Stefanidis, and Themis Palpanas. 2017. Parallel meta-blocking for scaling entity resolution over big heterogeneous data. *Inf. Syst.* 65 (2017), 137–157.
[8] Tom Heath and Christian Bizer. 2011. Linked Data: Evolving the Web into a Global Data Space. Morgan & Claypool Publishers.
[9] Stanley Hillner and Axel-Cyrille Ngonga Ngomo. 2011. Parallelizing LIMES for large-scale link discovery. In *I-SEMANTICS*. 9–16.
[10] Robert Isele, Anja Jentzsch, and Christian Bizer. 2011. Efficient Multidimensional Blocking for Link Discovery without losing Recall. In *WebDB*.
[11] Anja Jentzsch, Robert Isele, and Christian Bizer. 2010. Silk - Generating RDF Links While Publishing or Consuming Linked Data. In *ISWC*. 53–56.
[12] Hideki Kawai, Hector Garcia-Molina, Omar Benjelloun, David Menestrina, Euijong Whang, and Heng Gong. 2006. *P-Swoosh: Parallel Algorithm for Generic Entity Resolution*. Technical Report 2006-19.
[13] Hung-sik Kim and Dongwon Lee. 2007. Parallel linkage. In *CIKM*. 283–292.
[14] Lars Kolb, Andreas Thor, and Erhard Rahm. 2012. Dedoop: Efficient Deduplication with Hadoop. *PVLDB* 5, 12 (2012), 1878–1881.
[15] Lars Kolb, Andreas Thor, and Erhard Rahm. 2012. Multi-pass sorted neighborhood blocking with MapReduce. *Computer Science - R&D* 27, 1 (2012), 45–63.
[16] Axel-Cyrille Ngonga Ngomo. 2012. On Link Discovery using a Hybrid Approach. *J. Data Semantics* 1, 4 (2012), 203–217.
[17] Axel-Cyrille Ngonga Ngomo and Sören Auer. 2011. LIMES - A Time-Efficient Approach for Large-Scale Link Discovery on the Web of Data. In *IJCAI*. 2312–2317.
[18] Axel-Cyrille Ngonga Ngomo, Lars Kolb, Norman Heino, Michael Hartung, Sören Auer, and Erhard Rahm. 2013. When to Reach for the Cloud: Using Parallel Hardware for Link Discovery. In *ESWC*. 275–289.
[19] George Papadakis, George Alexiou, George Papastefanatos, and Georgia Koutrika. 2015. Schema-agnostic vs Schema-based Configurations for Blocking Methods on Homogeneous Data. *PVLDB* 9, 4 (2015), 312–323.
[20] George Papadakis, Georgia Koutrika, Themis Palpanas, and Wolfgang Nejdl. 2014. Meta-Blocking: Taking Entity Resolution to the Next Level. *IEEE Trans. Knowl. Data Eng.* 26, 8 (2014), 1946–1960.
[21] George Papadakis, George Papastefanatos, Themis Palpanas, and Manolis Koubarakis. 2016. Scaling Entity Resolution to Large, Heterogeneous Data with Enhanced Meta-blocking. In *EDBT*. 221–232.
[22] George Papadakis, Jonathan Svirsky, Avigdor Gal, and Themis Palpanas. 2016. Comparative Analysis of Approximate Blocking Techniques for Entity Resolution. *PVLDB* 9, 9 (2016), 684–695.
[23] George Papadakis, Leonidas Tsekouras, Emmanouil Thanos, George Giannakopoulos, Themis Palpanas, and Manolis Koubarakis. 2017. JedAI: The Force behind Entity Resolution. In *ESWC (demo paper)*.
[24] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. 2006. *Java concurrency in practice*. Pearson Education.
[25] Max Schmachtenberg, Christian Bizer, and Heiko Paulheim. 2014. Adoption of the Linked Data Best Practices in Different Topical Domains. In *ISWC*. 245–260.
[26] Giovanni Simonini, Sonia Bergamaschi, and H. V. Jagadish. 2016. BLAST: a Loosely Schema-aware Meta-blocking Approach for Entity Resolution. *PVLDB* 9, 12 (2016), 1173–1184.
[27] Dezhao Song and Jeff Heflin. 2011. Automatically Generating Data Linkages Using a Domain-Independent Candidate Selection Approach. In *ISWC*. 649–664.