Eliminating the Redundancy in Blocking-based Entity Resolution Methods

George Papadakis[$,♯], Ekaterini Ioannou[◇],
Claudia Niederée[♯], Themis Palpanas[‡], and Wolfgang Nejdl[♯]

[$] National Technical University of Athens, Greece gpapadis@mail.ntua.gr
[◇] Technical University of Crete, Greece ioannou@softnet.tuc.gr
[♯] L3S Research Center, Germany {surname}@L3S.de
[‡] University of Trento, Italy themis@disi.unitn.eu

ABSTRACT

Entity resolution is the task of identifying entities that refer to the same real-world object. It has important applications in the context of digital libraries, such as citation matching and author disambiguation. Blocking is an established methodology for efficiently addressing this problem; it clusters similar entities together, and compares solely entities inside each cluster. In order to effectively deal with the current large, noisy and heterogeneous data collections, novel blocking methods that rely on redundancy have been introduced: they associate each entity with multiple blocks in order to increase recall, thus increasing the computational cost, as well.

In this paper, we introduce novel techniques that remove the superfluous comparisons from any redundancy-based blocking method. They improve the time-efficiency of the latter without any impact on the end result. We present the optimal solution to this problem that discards all redundant comparisons at the cost of quadratic space complexity. For applications with space limitations, we also present an alternative, lightweight solution that operates at the abstract level of blocks in order to discard a significant part of the redundant comparisons. We evaluate our techniques on two large, real-world data sets and verify the significant improvements they convey when integrated into existing blocking methods.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: Information filtering

General Terms

Algorithms, Experimentation, Performance

Keywords

Data Cleaning, Entity Resolution, Redundancy-based Blocking

1. INTRODUCTION

Nowadays, the growing availability of semi-structured and structured data in the Web of Data opens new opportunities for digital libraries. These data collections can clearly profit from a variety of digital library principles and technologies, such as the systematic and uniform description of data by metadata, metadata harvesting services and technologies for federated search. Furthermore, they can be exploited to create new types of services by combining them with traditional types of library content. The integration of related data in meaningful ways relies on the detection of data records (from different collections) that refer to the same object, e.g., author.

The process of identifying, among a set of entities, those referring to the same real-world object is called *Entity Resolution* (**ER**). There are two main applications of this process in digital libraries: *citation matching* for identifying references that describe the same publication, and *author disambiguation* for identifying author profiles that pertain to the same person [9, 10, 25, 27]. The latter consists of detecting - within a collection of bibliographical records - the correct coupling between author names and persons, by resolving the *mixed citation problem* (same name - different persons) and the *split citation problem* (same person - different names) [14].

At its core, ER constitutes a quadratic problem, as each entity has to be compared with all others. To enhance its efficiency, blocking methods are typically employed [6, 11, 15]; they extract from every entity profile (or record) a Blocking Key Value (BKV) that encapsulates its most distinguishing information and define blocks on the equality (or similarity) of BKVs. Thus, each block corresponds to a specific instance of the BKV and contains all entities associated with that value.

However, in order to select the most reliable and distinguishing attributes of the given entity profiles, traditional blocking methods rely on a predefined entity schema. This renders them inapplicable for the Web of Data, due to the special characteristics of the latter: it involves individual collections that are highly heterogeneous, stemming from a rich diversity of sources, which evolve autonomously, following an unprecedented growth rate (especially the user-generated data of the Social Web, and the data created by sensors). More specifically, the following challenges are present in these settings:

Loose schema binding. The schemata describing entities may range from locally defined attributes to pure tag-style annotations, and data often have no strict binding to the employed schemata.

Noisy, missing, and inconsistent values. They are introduced in the data due to extraction errors, sources of low quality, and use of alternative descriptions. As a result, entity profiles may contain deficient, or even false information.

Extreme levels of heterogeneity. This is caused by the fact that data stem from a variety of distributed, self-organized, collaborative sources. Actually, heterogeneity pertains not only to schemata describing the same entity types, but also to profiles describing the same entity. For instance, Google

Base[1] encompasses 100,000 distinct schemata corresponding to 10,000 entity types [16].

High growth rates in terms of volume and fast evolution. This is caused partly due to automatic generation and partly due to the high involvement of users: they typically add new content, and update incorrect, outdated, or simply irrelevant information.

These inherent characteristics of heterogeneous information spaces break the fundamental assumptions of traditional blocking techniques. Novel blocking schemes, that do not require a predefined schema, have been introduced to effectively deal with these challenges. They all rely on *redundancy*, associating each entity with multiple blocks [3, 20, 21, 28]. In this way, they minimize the likelihood that two duplicate entities have no block in common, and achieve high levels of effectiveness (i.e., detected duplicate entities). This comes, though, at the cost of time efficiency: the resulting blocks are overlapping, and the same pairs of entities may be compared multiple times. Therefore, the main challenge for improving the efficiency of redundancy-based blocking methods is to eliminate the superfluous comparisons they entail, without affecting their accuracy.

In this paper, we address the above problem through an abstraction of the redundancy-based blocking techniques: blocks are associated with an index indicating their position in the processing list and entities are associated with a list of the indices of the blocks that contain it. Thus, we can identify whether a pair of entities contained in the current block has already been compared in another block simply by comparing their least common block index with the index of the current block. In this way, we achieve the optimal solution to the problem, since we efficiently propagate all executed comparisons, without explicitly storing them.

The above approach has low computational cost, but results in quadratic space complexity. In order to remedy this drawback, we introduce a method that approximates the optimal solution by gracefully trading space for computational cost. It comprises of a series of block manipulation techniques, which discard those blocks that exclusively entail superfluous comparisons (i.e., they are entirely contained in another block), and merge pairs of highly overlapping blocks, giving birth to blocks that entail less comparisons. These functionalities are facilitated by mapping the blocks to a Cartesian space and contrasting their spatial representations, without the need to examine their contents analytically.

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

(1) We formulate the problem of purging redundant comparisons from a blocking technique and explain how its solution can be facilitated through the abstraction of blocks (i.e., enumerating and mapping them to the Cartesian space).

(2) We describe Comparisons Propagation, an optimal solution to this problem, which efficiently propagates all executed comparisons based on the enumeration of blocks. This method is suitable for applications that can afford high space complexity.

(3) We further propose a solution that partially discards redundant comparisons, trading space requirements for time complexity. It consists of a series of methods that remove blocks involving exclusively redundant comparisons and merge highly overlapping ones.

(4) Finally, we thoroughly evaluate our methods on two large, real-world data sets, demonstrating the great benefits they convey to the efficiency of existing blocking methods.

The rest of the paper is structured as follows. Section 2 summarizes previous work and Section 3 defines the basic notions of our algorithms. Section 4 introduces our approach to determining the processing order of blocks and mapping them to the Cartesian Space, and Section 5 presents our approach to propagating comparisons and manipulating blocks. Experimental evaluation is presented in Section 6, while Section 7 concludes the paper.

[1]See `http://www.google.com/base`.

2. RELATED WORK

A variety of methods for solving the ER problem has been presented in literature. They range from string similarity metrics [2], to similarity methods using transformations [19, 26] and relationships between data [5, 12]. A comprehensive overview of the existing work in this domain can be found in [4, 6, 13].

The approximate methods of data blocking typically associate each record (i.e., entity) with a *Blocking Key Value* (BKV). They define blocks on the equality (or similarity) of BKVs and compare solely the entities that are contained in the same block [6]. For instance, the Sorted Neighborhood approach [11] orders records according to their BKV and slides a window of fixed size over them, comparing the records it contains. The StringMap method [15] maps the BKV of each record to a multi-dimensional Euclidean space, and employs suitable data structures for efficiently identifying pairs of similar records. The q-grams blocking approach [8] builds overlapping clusters of records that share at least one q-gram (i.e., sub-string of length q) of their BKV. Canopy clustering [17] employs a cheap string similarity metric for building high-dimensional overlapping blocks, whereas the Suffix Arrays approach [3] considers the suffixes of the BKV instead. [28] explores another aspect of these blocking approaches, arguing that more duplicates can be detected and more pair-wise comparisons can be saved through the iterative distribution of identified matches to subsequently (re-)processed blocks.

The performance of blocking methods typically depends on the fine-tuning of a wealth of application- and data-specific parameters [3, 29]. To automate the parameter setting procedure, several methods that model it as a machine learning problem have been proposed in the literature. For instance, [18] defines it as learning disjunctive sets of conjunctions that consist of an attribute (used for blocking) and a method (used for comparing the corresponding values). Similarly, [1] considers disjunctions of *blocking predicates* (i.e., conjunctions of attributes and methods) along with predicates combined in disjunctive normal form (DNF). On the other hand, [29] introduces a method for adaptively and dynamically setting the size of the sliding window of the Sorted Neighborhood approach.

Attribute-agnostic blocking methods were recently introduced to make blocking applicable to voluminous, heterogeneous data collections, such as the Web of Data. These methods do not need a predefined schema for grouping entities into blocks, as they completely disregard attribute names. In this way, they are able to handle thousands of attribute names without requiring the fine-tuning of numerous parameters. Instead, they tokenize (on all special characters) the attribute values of each entity profile, and create an individual block for each token; that is, every block corresponds to a specific token and contains all entities having this token in their profile [21]. Blocks of such low-level granularity guarantee high effectiveness due to the high redundancy they convey: each entity is associated with multiple blocks, which are, thus, overlapping. Hence, the likelihood of a missed match (i.e., a pair of duplicates that has no block in common) is low. This redundancy-based approach is a common practice among blocking techniques for noisy, but homogeneous data, as well [3, 17, 20, 28].

To the best of our knowledge, this is the first work on formally defining and dealing with the problem of eliminating redundant comparisons of blocking methods for ER.

3. PROBLEM DEFINITION

To formally describe the problem we are tackling in this paper, we adopt the definitions introduced in [21] for modeling entity profiles and entity collections. As such, an **entity profile p** is a tuple $\langle id, A_p \rangle$, where A_p is a set of attributes a_i, and $id \in \mathcal{ID}$ is a global identifier for the profile. Each **attribute $\mathbf{a_i}$** $\in A_p$ is a tuple $\langle n_i, v_i \rangle$, consisting of an **attribute name $\mathbf{n_i}$** and an **attribute value $\mathbf{v_i}$**. Each attribute value can also be an identifier, which allows for modeling relationships between entities. An **entity collection** $\mathcal{E}$ is a tuple $\langle A_E, V_E, ID_E, P_E \rangle$, where A_E is the set of attribute names appearing in it, V_E is the set of values used in it, $ID_E \subseteq \mathcal{ID}$ is the set of global identifiers contained in it, and $P_E \subseteq ID_E$ is the set of entity profiles that it comprises. We define a blocking scheme as follows:

Definition 1. *A **blocking scheme bs** for an entity collection $\mathcal{E}$ is defined by a transformation function $f_t : \mathcal{E} \mapsto T$ and a set of constraint functions $f^i_{cond} : T \times T \mapsto \{true, false\}$. The **transformation function $\mathbf{f_t}$** derives the appropriate blocking representation from the complete entity profile (or parts of it). The **constraint function $\mathbf{f^i_{cond}}$** is a transitive and symmetric function that encapsulates the condition that has to be satisfied by two entities, if they are to be placed in the same block b_i.*

Apparently, any blocking method can define and use its own blocking scheme that follows the above definition. For example, the schemes described in Section 2 consist of a transformation function that extracts the BKV from an entity profile and a set of constraint functions that define blocks on the equality (or similarity) of the BKVs. Once a blocking scheme is applied on an entity collection, a set of blocks is derived, whose instances are formally defined as follows:

Definition 2. *Given an entity collection $\mathcal{E}$ and a blocking scheme bs for $\mathcal{E}$, a **block $\mathbf{b_i}$** $\in B$ is the maximal subset, with a minimum cardinality of 2, that is defined by the transformation function f_t and one of the constraint functions f^i_{cond} of bs:*

$$b_i \subseteq \mathcal{E} \wedge \forall p_1, p_2 \in \mathcal{E} : f^i_{cond}(f_t(p_1), f_t(p_2)) = true \Rightarrow p_1, p_2 \in b_i.$$

The ER process on top of a blocking method consists of iterating over its set of blocks B in order to compare the entities contained in each one of them. We use $\mathbf{m_{i,j}}$ to denote a match between two profiles p_i and p_j that have been identified as matching $p_i \equiv p_j$ (i.e., describing the same real-world object). The output, therefore, of a blocking method is a set of matches, which we denote as $\mathbf{M}$.

To address the aforementioned characteristics of heterogeneous information spaces, *redundancy-bearing* blocking methods have been recently introduced [3, 20, 21, 28]. They associate each entity with multiple, overlapping blocks. This practice minimizes the likelihood that two duplicate entities have no block in common, thus resulting in higher effectiveness for the ER process. Efficiency, on the other hand, is significantly downgraded, due to the redundant comparisons between pairs of entities that appear in many blocks. Apparently, the higher the redundancy conveyed by a blocking method, the lower the efficiency of the ER process.

In this paper, we focus on developing methods that enhance the efficiency of redundancy-based blocking methods, *without* affecting their effectiveness. To this end, our techniques aim at eliminating the superfluous comparisons of redundancy-bearing blocking methods in order to save considerable computational effort. In this way, they can operate on top of any blocking method, without altering its effectiveness, producing an output that is equivalent to the original one. The following definition introduces the concept of *semantically equivalent blocking sets*:

Definition 3. *A blocking set B' is **semantically equivalent** to blocking set B, if the set of matches resulting from blocking set B' are equal to the set of matches resulting from blocking set B (i.e., $\mathcal{M}_{\mathcal{B}'} = \mathcal{M}_{\mathcal{B}}$).*

Based on the above definition, we now formally state the problem we are addressing in this paper:

Problem 1. *Given a set of blocks B that are derived from a redundancy-bearing blocking technique, find the semantically equivalent blocking set B' that involves no redundant pair-wise comparisons.*

4. BLOCK SCHEDULING AND MAPPING

As stated above, our goal is to propose generic methods for enhancing the efficiency of any redundancy-bearing blocking technique (such as the ones discussed in Section 2). Therefore, the methods we describe make no assumptions on the mechanism or functionality of the underlying blocking method. Instead, they treat blocks at an abstract level, considering solely the identifiers of the entities they contain (i.e., each block is represented as a set of entity ids).

We distinguish between two types of blocks according to the lineage of their entities. The first type of blocks is called *unilateral*, since it contains entities of the same lineage, i.e., stemming from the same entity collection. This type of blocks arises when integrating one dirty collection (i.e., a collection that contains duplicate entities) either with a clean, duplicate-free collection (i.e., *Dirty-Clean*), or with another dirty collection (i.e., *Dirty-Dirty*). Both cases are equivalent with resolving a single, dirty entity collection, where each entity profile could match to any other [24]. More formally, the blocks of this kind are defined as follows:

Definition 4. *A **unilateral block** is a block containing entity ids from a single entity collection $\mathcal{E}$, thus being of the form $b_i = \{id_1, id_2, \ldots, id_n\}$, where $id_i \in \mathcal{ID}$.*

The second type of blocks is called *bilateral*, and arises when integrating two individually clean entity collections, $\mathcal{E}_1$ and $\mathcal{E}_2$, that are overlapping (*Clean-Clean*) [24]. The goal is, therefore, to identify matches only between $\mathcal{E}_1$ and $\mathcal{E}_2$, thus requiring that each block contains entities from both input collections. More formally, this kind of blocks is defined as follows:

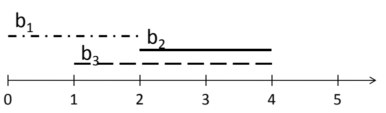
Definition 5. *A **bilateral block** is a block containing entity ids from two entity collections, $\mathcal{E}_1$ and $\mathcal{E}_2$. It follows the form $b_{i,j} = \{\{id_{i,1}, id_{i,2}, \ldots, id_{i,n}\}, \{id_{j,1}, id_{j,2}, \ldots, id_{j,m}\}\}$, where $id_{i,k} \in \mathcal{ID}_1$ and $id_{j,l} \in \mathcal{ID}_2$. The subsets $b_i = \{id_{i,1}, id_{i,2}, \ldots, id_{i,n}\}$ and $b_j = \{id_{j,1}, id_{j,2}, \ldots, id_{j,m}\}$ are called the **inner blocks** of $b_{i,j}$.*

4.1 Block Scheduling

Specifying the processing order of blocks is important for the effectiveness of ER techniques. This order forms the basis for **block enumeration**, which associates each block with an integer that represents its position in the processing list. This practice finds application in various techniques, such as the propagation of comparisons (see Section 5.1). The processing order is also an integral part of *lossy efficiency techniques*, like Block Pruning in [21]; these are methods that sacrifice (to some extent) the effectiveness of a blocking method in order to enhance its efficiency. They do so by discarding comparisons according to some criteria, even if they involve non-redundant comparisons among matching entities.

Scheduling techniques typically associate each element of the given set of blocks B with a numerical value and sort B in ascending or descending order of this value. Their computational cost

(a)



(b)

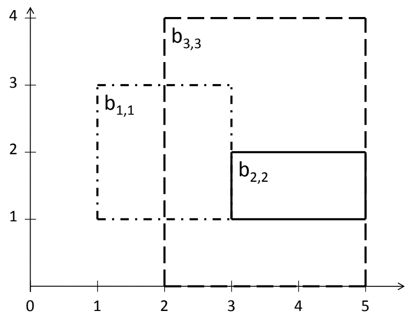


Figure 1: Illustration of block mapping.

is $O(|B| \cdot \log |B|)$, which scales even for large sets of blocks. In each case, the most suitable approach for determining the processing order of blocks depends heavily on the application at hand. For the needs of the methods we introduce in Section 5, we define a different scheduling method for each kind of block. In particular, unilateral blocks are ordered in ascending order of cardinality: the more entities a block b_i contains, the higher its position in the list. Bilateral blocks, on the other hand, are ordered in ascending order of their *utility* [21]: $u_{b_{i,j}} = \frac{1}{max(|b_i|,|b_j|)}$, where $|b_i|$ and $|b_j|$ are the cardinalities of the inner blocks of the bilateral block $b_{i,j}$. Bilateral blocks of equal utility are ordered in ascending order of the cardinality of their smallest inner block.

4.2 Block Mapping

We now introduce our approach for Block Mapping. The gist of this technique is that it allows us to efficiently check whether two blocks have overlapping content (i.e., they share some entities), without exhaustively comparing them. The mapping is performed by transforming blocks into the Cartesian space; for unilateral blocks this corresponds to Cartesian coordinates in one dimension (i.e., lines), and for bilateral blocks to coordinates in two dimensions (i.e., rectangles). Thus, Block Mapping is performed by assigning each entity to a point on the corresponding axis.

Example 1. *Figure 1(a) illustrates the mapping of the unilateral blocks* $b_1 = \{id_2, id_3, id_4\}$, $b_2 = \{id_0, id_1, id_4\}$, *and* $b_3 = \{id_0, id_1, id_3, id_4\}$ *on the X-axis. Their entities are assigned to coordinates as follows: C={⟨id_0, 3⟩, ⟨id_1, 4⟩, ⟨id_2, 0⟩, ⟨id_3, 1⟩, ⟨id_4, 2⟩}.*
Figure 1(b) illustrates the mapping of the bilateral blocks $b_{1,1} = \{\{id_{1,0}, id_{1,2}\}, \{id_{2,0}, id_{2,1}\}\}$, $b_{2,2} = \{\{id_{1,0}, id_{1,3}\}, \{id_{2,1}, id_{2,3}\}\}$ *and* $b_{3,3} = \{\{id_{1,0}, id_{1,3}, id_{1,4}\}, \{id_{2,1}, id_{2,2}, id_{2,4}\}\}$ *to the XY-axes, where* $id_{1,i} \in \mathcal{E}_1$ *and* $id_{2,i} \in \mathcal{E}_2$. *The entities of* $\mathcal{E}_1$ *are transformed to points on the X-axis as follows:* C_X*={⟨$id_{1,0}$, 3⟩,⟨$id_{1,1}$, 4⟩, ⟨$id_{1,2}$,1⟩, ⟨$id_{1,3}$, 5⟩, ⟨$id_{1,4}$, 2⟩}, whereas the entities of* $\mathcal{E}_2$ *are mapped to points on the Y-axis as follows:* C_Y*={⟨$id_{2,0}$, 3⟩,⟨$id_{2,1}$, 1⟩,⟨$id_{2,2}$,0⟩, ⟨$id_{2,3}$, 2⟩, ⟨$id_{2,4}$, 4⟩}.*

The difference between the size b_i^{sp} of the spatial representation of a block b_i and its actual size b_i^{as} is called **spatial deviation** sp_i of block b_i. More formally, it is defined as follows:

$$sp_i = b_i^{sp} - b_i^{as},$$

Algorithm 1: Mapping Blocks to the Cartesian Space.

```
Input:  B={b_i}        a set of unilateral blocks
Output: C={⟨id_i, j⟩} a mapping of entity ids to coordinates
1  B′ ← blockScheduling(B);
2  C ← {};
3  lastIndex ← 0;
4  foreach b_i ∈ B′ do
5      E ← sortInAscendingOrderOfFrequency(b_i.entities());
6      foreach e ∈ E do
7          if (!C.containsKey(e.id)) then
8              C ← C ∪ {⟨e.id, lastIndex⟩};
9              lastIndex++;
10 return C;
```

where b_i^{sp} is the length (area) of the spatial representation of a unilateral (bilateral) block b_i, and b_i^{as} is the actual length (area) of the unilateral (bilateral) block b_i. In the case of unilateral blocks, we have $b_i^{as} = |b_i| - 1$, while for a bilateral block $b_{i,j}$ it is equal to $b_{i,j}^{as} = (|b_i| - 1) \cdot (|b_j| - 1)$. For example, $b_{2,2}$ has an actual area of ((2-1)×(2-1)=)1, whereas its spatial representation has an area of ((5-3)×(2-1)=)2; that is, $sp_{2,2} = 1$.

The value of sp_i is always positive, but in the ideal case it should be equal to 0. This requirement can be easily satisfied for non-overlapping blocks, by associating the entities of each block with contiguous coordinates. In the case of overlapping blocks, though, the spatial transformation leads to a positive deviation, since it cannot be done independently for each block: assigning a coordinate to an entity id_i in the context of a block b_i can be contiguous with the rest of entities in b_i, but not necessarily with the other entities that share blocks with id_i.

Example 2. *Consider the entities in Figure 1(a). The way the depicted blocks are mapped is the optimal one, since the entities of every block are contiguous. Imagine, though, that we place an additional entity to each block:* id_5 *to* b_1, id_6 *to* b_2 *and* id_7 *to* b_3. *In this case, there is no way of mapping the new blocks to the X-axis, so that the entities of each block are contiguous.*

The above discussion gives rise to the following optimization problem:

Problem 2. *Given a set of blocks B, transform its elements to the Cartesian space, so that their aggregated spatial deviation* $\sum_{b_i \in B}(b_i^{sp} - b_i^{as})$ *is minimized.*

In our methods, we require that emphasis is placed on minimizing the spatial deviation of large blocks (we elaborate on the reasons in Section 5.2.2). That is, *the larger a block is, the lower its spatial deviation should be*. More formally, this optimization problem can be defined as the minimization of the following quantity:

$$\sum_{b_i \in B} b_i.size() \times (b_i^{sp} - b_i^{as}), \quad (1)$$

where $b_i.size()$ is the **size of block** b_i. For a unilateral block, it is equal to its cardinality (i.e., number of entities it contains), while for a bilateral block $b_{i,j}$ it is equal to the sum of cardinalities of its inner blocks: $b_{i,j}.size() = |b_i| + |b_j|$.

We solve this optimization problem using a scalable method, applicable to voluminous data collections. Algorithm 1 outlines this method for the case of unilateral blocks; for bilateral ones, the algorithm is applied twice, independently for each axis, considering in each iteration solely the corresponding entity collection. In essence, the algorithm assigns coordinates from the interval $[0, |\mathcal{E}| - 1]$ to the entity profiles of the given collection $\mathcal{E}$.

After Block Scheduling, it starts assigning the entities of the last (usually largest) block to contiguous coordinates, thus minimizing the spatial deviation of this block. To ensure the minimal spatial deviation for the rest of the blocks, as well, the profiles are ordered and mapped in ascending order of their *frequency* (i.e., the number of blocks associated with each entity): the least frequent of the not-yet-mapped entities takes the first available coordinate, the second least frequent takes the next coordinate etc. Two entities that share many blocks are more likely to be contiguous in this way. The algorithm is then repeated for the remaining blocks, traversing their ordered list from bottom to top. The algorithm has a linear space complexity, $O(|B| + |\mathcal{E}|)$, and time complexity of $O(|B| \cdot \log |B| + |\mathcal{E} \cdot \log |\mathcal{E}|)$, due to the sorting of blocks and entities.

Example 3. *The result of applying this algorithm is illustrated in Figure 1(a). The profiles of b_1 are mapped to the X axis as follows: id_2 has frequency 0 and goes to the first available coordinate (i.e., 0), id_3 with frequency 1 goes to next available coordinate (i.e., 1) , and, finally, id_4 with frequency 2 goes to point 2.*

5. APPROACH

In this section, we present the methods we developed for reducing the redundancy of blocking methods (Problem 1), based on the Block Scheduling and Mapping techniques we introduced above. The optimal solution to this problem (i.e., the one that discards all redundant comparisons) is presented in Section 5.1. Its effectiveness, though, comes at the cost of high space complexity, caused by the data structure it employs. As an alternative, we present in Section 5.2 an approximate solution, that removes the high space requirements.

5.1 Comparisons Propagation

Block Scheduling determines the processing order of blocks, and enables their enumeration; that is, each block is assigned to an index indicating its position in the processing list. Based on this enumeration, the propagation of comparisons is made feasible through a common data structure, namely a *hash table*; in particular, its *keys* are the ids of the entities of a given collection $\mathcal{E}$, and its *values* are lists of the indices of the blocks that contain the corresponding entities. The elements of these lists are sorted in ascending order, from the lowest block index to the highest.

This data structure can be used in the context of a blocking method in the following way: to compare a pair of entities, the **Least Common Block Index Condition** should be satisfied. That is, the lowest common block index of these entities should be equal to the index of the current block, indicating in this way that this is the first block in the processing list that contains both of them. Otherwise, if the former index is lower than the latter, the entities have already been compared in another block, and the comparison is redundant. In this way, each pair of entities is compared just once, and Comparisons Propagation provides the optimal solution to Problem 1.

Theorem 1 (Optimality of Comparisons Propagation). *Given a set of blocks B, Comparisons Propagation produces the semantically equivalent set of blocks B′ that entails no redundant pair-wise comparisons.*

Proof. Let us assume that the set of blocks produced by Comparisons Propagation entails redundant comparisons. This means that there is a blocking set B'' that is semantically equal to B and involves no redundant comparisons. Hence, there must be at least one pair of entities that is compared twice in B' and just once in B''. The Least Common Block Index Condition is, therefore, satisfied in two blocks of B', which is a contradiction. Thus, $B'' \equiv B'$. ∎

Algorithm 2: Propagating Comparisons.

```
Input:  B a set of blocks
Output: B'' the semantically equivalent set of blocks with no
        redundant comparisons
1  $B \leftarrow$ blockScheduling($B$);
2  $B' \leftarrow$ blockEnumeration($B$);
3  $B'' \leftarrow \{\}$;
4  entityIndex $\leftarrow$ indexBlocksOnEntityIds($B'$);
5  foreach $b_i \in B'$ do
6      $E \leftarrow b_i$.entities();
7      for $i \leftarrow 1$ to $E.size$ do
8          $B_{E_i} \leftarrow$ entityIndex.associatedBlocks($E[i]$);
9          for $j \leftarrow i + 1$ to $E.size$ do
10             $B_{E_j} \leftarrow$ entityIndex.associatedBlocks($E[j]$);
11             if ($b_i$.index = leastCommonBlockIndex($B_{E_i}$,$B_{E_j}$)) then
12                 $b_i \leftarrow$ newBlock($E[i]$, $[E[j]]$);
13                 $B'' \leftarrow B'' \bigcup b_i$ ;
14 return $B''$;
```

Algorithm 2 outlines the way Comparisons Propagation operates on a set of blocks B in order to produce its semantically equivalent set of blocks B' that is free of redundant comparisons. In essence, B' consists of blocks with minimum cardinality, since each non-redundant comparison results in a new block that contains the corresponding pair of entities. This may result in a very large number of blocks, and storing them poses a serious challenge. Processing them on-the-fly, though, is an efficient alternative.

Comparisons Propagation can be integrated in the execution of any blocking method, without affecting its time complexity. The reason is that the computation of the least common block index is linear to the number of blocks associated with the corresponding pair of entities (due to the ordering of indices in the values of the hash table). Its space complexity, though, is equal to $O(|B| \cdot |\mathcal{E}|)$, in the worst case (i.e., each entity is placed in all blocks), where $|B|$ is the total number of blocks, and $|\mathcal{E}|$ is the cardinality of the given entity collection (for Clean-Clean ER, this cardinality is equal to $|\mathcal{E}_1| + |\mathcal{E}_2|$). In practice, however, space complexity depends on the level of redundancy introduced by the underlying blocking method. In fact, it is equal to $O(|\mathcal{E}| \cdot B\bar{P}E)$, where $B\bar{P}E$ is an estimate of redundancy, denoting the average number of blocks per entity.

5.2 Block Manipulation

Block Manipulation consists of a series of techniques that operate on two levels: first, they investigate the given set of blocks in order to discard those elements that contain purely redundant comparisons. In this way, they reduce not only the number of comparisons, but also the number of blocks that will be processed in the next level. Second, they aim at identifying profitable block merges; that is, pairs of highly overlapping blocks, which, when combined, result in a block with fewer comparisons. The combined result of these two levels approximates the optimal solution of Comparisons Propagation at a lower space complexity. The individual strategies of Block Manipulation are analytically presented in the following paragraphs, in the order they should be executed.

5.2.1 Block Cleaning

Cleaning a set of blocks B is the process of purging the duplicate elements from it. These are blocks that contain exactly the same entities with another block, regardless of the constraint function defining each of them (i.e., independently of the information that is associated with them). We call such blocks identical, and, depending on their lineage, we formally define them as follows:

Algorithm 3: Mining a clean set of blocks.

```
Input: $B$ a clean set of highly similar blocks
Output: $B''$ the semantically equivalent, mined set of blocks
1  $B' \leftarrow$ blockScheduling($B$);
2  $DominatedB \leftarrow \{\}$;
3  for $i \leftarrow 1$ to $B'.size$ do
4      for $j \leftarrow B'.size$ to $i+1$ do
5          if ($B'[i].size() \neq B'[j].size()$) then
6              break;
7          if (areaConditionHolds($B'[i]$, $B'[j]$)) then
8              if (isDominated($B'[i]$, $B'[j]$)) then
9                  $DominatedB \leftarrow DominatedB \cup B'[i]$;
10                 break;
11 $B'' \leftarrow B'$ - $DominatedBlocks$;
12 return $B''$;
```

Definition 6. *Given a set of unilateral blocks B, a block $b_i \in B$ is* ***unilaterally identical*** *with another block $b_j \in B$, denoted by* $\mathbf{b_i} \equiv \mathbf{b_j}$, *if both blocks contain the same entities, regardless of their constraint functions, f^i_{cond} and f^j_{cond}: $b_i \equiv b_j \Leftrightarrow b_i \subseteq b_j \wedge b_j \subseteq b_i$.*

Definition 7. *Given a set of bilateral blocks B, a block $b_{i,j} \in B$ is* ***bilaterally identical*** *with another block $b_{k,l} \in B$, denoted by* $\mathbf{b_{i,j}} \equiv \mathbf{b_{k,l}}$, *if their corresponding inner blocks are unilaterally identical: $b_{i,j} \equiv b_{k,l} \Leftrightarrow b_i \equiv b_k \wedge b_j \equiv b_l$.*

In this context, the process of Block Cleaning can be formally defined as follows:

Problem 3 (Block Cleaning). *Given a set of blocks B, reduce B to its semantically equivalent subset $B' \subseteq B$ that contains no identical blocks. We call B' a* ***clean set of blocks****.*

The solution to this problem can be easily implemented by associating each block with a *hash signature*; its value is equal to the sum of the coordinates assigned to its entities by Block Mapping. Identical blocks necessarily have the same signature, but not vice versa: signature equality can also lead to false positives. For this reason, the size as well as the elements of two blocks with the same signature are analytically compared to make sure that they are indeed identical. In practice, this functionality is efficiently offered by default by most programming languages. Both its time and space complexity are linear to the size of the input set of blocks (i.e., $O(|B|)$), as it involves traversing its elements just once.

5.2.2 Block Mining

Given a clean set of blocks, the process of mining it consists of identifying the blocks that are subsets of at least one other block in the set; that is, blocks whose entities are all contained in some other block, independently of the corresponding constraint functions. This situation is called a *relation of dominance*, where the latter is the *dominant* block, and the former the *dominated* one. This is more formally defined as follows:

Definition 8. *Given a clean set of unilateral blocks B, a block $b_i \in B$ is* ***unilaterally dominated*** *by another block $b_j \in B$, denoted by* $\mathbf{b_i} \preceq \mathbf{b_j}$, *if b_i is a proper subset of b_j, regardless of the constraint functions f^i_{cond} and f^j_{cond}: $b_i \preceq b_j \Leftrightarrow |b_i| < |b_j| \wedge \nexists id_i \in b_i : id_i \notin b_j$.*

Definition 9. *Given a clean set of bilateral blocks B, a block $b_{i,j} \in B$ is* ***bilaterally dominated*** *by another block $b_{k,l} \in B$, denoted by* $\mathbf{b_{i,j}} \preceq \mathbf{b_{k,l}}$, *if at least one inner block of $b_{i,j}$ is unilaterally dominated by the corresponding inner block of $b_{k,l}$ and the other is either unilaterally identical or unilaterally dominated: $b_{i,j} \preceq b_{k,l} \Leftrightarrow (|b_i| \preceq |b_k| \wedge |b_j| \preceq |b_l|) \bigvee (|b_i| \preceq |b_k| \wedge |b_j| \equiv |b_l|) \bigvee (|b_i| \equiv |b_k| \wedge |b_j| \preceq |b_l|)$.*

In this context, the problem of Block Mining can be formally defined as follows:

Problem 4 (Block Mining). *Given a clean set of block B, reduce B to its semantically equivalent subset $B' \subseteq B$ that contains no dominated blocks. We call B' a* ***mined set of blocks****.*

Apparently, this constitutes another quadratic problem, since the elements of every block have to be compared with those of all others. However, the abstract representation of blocks we are proposing leads to a series of necessary conditions that have to be satisfied by a pair of blocks, if one of them is dominated. The benefit is that these conditions can be checked in a fast and easy way, without the need to analytically compare the elements of the blocks. The conditions are also complementary, with their conjunction forming a composite mining method that effectively restricts the required number of comparisons. In the following, we describe them in more detail.

(i) ***Size Condition (SC)***. In a clean set of blocks B, there cannot be a relation of dominance among a pair of *equally sized* blocks. Instead, the dominant block has to be larger in size than the dominated one. Therefore, to check whether a block is dominated, we need to compare it solely with blocks of larger size.

(ii) ***Area Condition (AC)***. Block mapping adds an additional condition for a relation of dominance: the spatial representation of the dominated block has to be fully contained in the representation of the dominant one; that is, the line (area) of the former lies entirely inside the line (area) mapped to the latter. More formally, the AC for a unilateral block b_i to be dominated by a block b_j is expressed as follows:

$$b_i \preceq b_j \Rightarrow (min(b_j.entityCoods) \leq min(b_i.entityCoods) \\ \wedge max(b_i.entityCoods) \leq max(b_j.entityCoods)),$$

where $b_k.entityCoods$ is the set of coordinates assigned to the entities of block b_k. Similarly, the AC for a bilateral block $b_{i,j}$ to be dominated by a block $b_{k,l}$ takes the following form:

$$b_{i,j} \preceq b_{k,l} \Rightarrow (min(b_k.entityCoods) \leq min(b_i.entityCoods) \\ \wedge max(b_i.entityCoods) \leq max(b_k.entityCoods)) \\ \wedge (min(b_l.entityCoods) \leq min(b_j.entityCoods) \\ \wedge max(b_j.entityCoods) \leq max(b_l.entityCoods)).$$

AC is illustrated in Figure 1(a); b_1 is smaller than b_3, but cannot be dominated by it, since the line of b_1 is not entirely covered by the line of b_3. On the other hand, b_2 satisfies the AC with respect to b_3 but not with b_1.

As mentioned in Section 4.2, the priority of Algorithm 1 is to ensure that large blocks end up with low spatial deviation. The reason is that AC is intended to be used in conjunction with SC. In this way, the latter condition saves comparisons among equally sized blocks, even if their spatial representations are intersecting, while the former saves unnecessary comparisons that involve large blocks. Without AC, each block is inevitably compared with all smaller blocks, even if they share no entities.

(iii) ***Entity Condition (EC)***. Another, straightforward way of considering the content of blocks before comparing them is to consider merely those pairs of blocks that have at least one entity in common. This can be achieved by using the same data structure that is employed in Comparisons Propagation: a hash table that contains for each entity a list of the blocks that are associated with it. However, this solution has the same (quadratic) space complexity with Comparisons Propagation, thus violating the requirement for an alternative solution with minimal space requirements. Thus, we replace it with a near-duplicates detection method.

Locality Sensitive Hashing (**LSH**) [7] constitutes an established method for hashing items of a high-dimensional space in such a way that similar items (i.e., near duplicates) are assigned to the same hash value with a high probability p_1. In addition, dissimilar items are associated with the same hash values with a very low probability p_2. In our case, blocks are the items that are represented in the high-dimensional space of $\mathcal{E}$ (or $\mathcal{E}_1$ and $\mathcal{E}_2$) through Block Mapping. Thus, LSH can be employed to group highly similar blocks in buckets, so that it suffices it compare blocks contained in the same bucket. The details of the configuration of LSH we employ are presented in Section 6.

Algorithm 3 outlines the steps of our Block Mining algorithm. In short, it encompasses two nested loops, with SC and AC integrated in the inner one. Note that the input consists of the blocks contained in the same bucket of LSH. It is worth noting that the nested loop starts from the bottom of the ordered list of blocks and traverses it to the top. In this way, smaller blocks are first compared with the largest ones. The reason is that the larger the difference in the size of the two blocks, the higher the likelihood that the larger block contains all the elements of the smaller one (thus, dominating it). SC is encapsulated in lines 5 and 6, while AC in line 7. Note that SC terminates the inner loop as soon as an equally sized block is encountered, because Block Scheduling ensures that the next blocks are of equal or smaller size.

5.2.3 Block Merging

An effective way of discarding superfluous comparisons is to identify blocks that are highly overlapping. These are blocks that share so many entities that, if merged, they would result in fewer comparisons than the sum of the comparisons needed for each one. Merging such blocks eliminates their redundant comparisons, thus enhancing efficiency without any impact on effectiveness. Indeed, pairs of entities that are common among the original blocks (i.e., the source of redundancy) are considered only once in the new blocks, while the pairs of duplicate entities are maintained without any change. This situation is illustrated in the following example:

Example 4. *Consider the following unilateral blocks:* $b_1 = \{id_1, id_2, id_3, id_4, id_5\}$, $b_2 = \{id_1, id_2, id_4, id_5, id_6\}$ *and* $b_3 = \{id_1, id_3, id_4, id_5, id_7, id_8\}$. *Individually, these blocks involve 35 comparisons, in total. However, b_2 and b_3 contain most of the comparisons in b_1 (they share four entities with b_1). Merging b_1 with b_2 leads to the new block $b'_2 = \{id_1, id_2, id_3, id_4, id_5, id_6\}$. We now need 30 comparisons in total (between b'_2 and b_3). In addition, merging b'_2 with b_3 in a single block containing all 8 entities further reduces the total number of comparisons to 28 (i.e., 20% less than the initial number of comparisons).*

More formally, the merge of two unilateral/bilateral blocks is defined as follows:

Definition 10. *Given a set of unilateral blocks B, the* ***unilateral merge*** *of a block b_i with a block b_j is a new unilateral block* $\mathbf{b_{m_{i,j}}}$ *that contains the union of the entities of b_i and b_j:* $b_{m_{i,j}} = b_i \cup b_j$.

Definition 11. *Given a set of bilateral blocks B, the* ***bilateral merge*** *of a block $b_{i,j}$ with a block $b_{k,l}$ is a new bilateral block* $\mathbf{b_{m_{i,k},m_{j,l}}}$, *whose inner blocks constitute the unilateral merge of the corresponding inner blocks of $b_{i,j}$ and $b_{k,l}$:*
$b_{m_{i,k},m_{j,l}} = \{b_i \cup b_k, b_j \cup b_l\}$.

Algorithm 4: Merging a mined set of blocks.

```
   Input:  $B$ a clean, mined set of blocks
   Output: $B''$ the semantically equivalent, merged set of blocks
1  $B' \leftarrow$ blockMapping($B$);
2  $FromMergesB \leftarrow \{\}$;
3  $MergedB \leftarrow \{\}$;
4  $OpenB \leftarrow \{\}$;
5  $ProcessedB \leftarrow \{\}$;
6  for $i \leftarrow 0$ to $spatialBlocks.maxDimension$ do
7      $NewOpenB \leftarrow$ getBlocksStartingAtIndex($i$, $B'$);
8      $NewOpenB \leftarrow NewOpenB \bigcup FromMergesB$;
9      $NewOpenB \leftarrow NewOpenB \setminus MergedB$;
10     $NewOpenB \leftarrow NewOpenB \setminus OpenB$;
11     $NewOpenB' \leftarrow$ blockScheduling($NewOpenB$);
12     $OpenB \leftarrow$ blockScheduling($OpenB \bigcup NewOpenB$);
13     $FromMergesB \leftarrow \{\}$;
14     foreach $b_i \in NewOpenB'$ do
15         $mostSimilarBlock \leftarrow$ getMostSimilarBlock($b_i$, $OpenB$);
16         if $mostSimilarBlock \neq null$ then
17             $newBlock \leftarrow$ mergeBlocks($b_i$, $mostSimilarBlock$);
18             $newBlock$.mapToCartesianSpace();
               $FromMergesB$.add($newBlock$);
19             $MergedB$.add($b_i$);
20             $MergedB$.add($mostSimilarBlock$);
21             $OpenB$.remove($b_i$);
22             $OpenB$.remove($mostSimilarBlock$);
23     $OpenB$.addAll($NewOpenB$);
24     $EndingBlocks \leftarrow$ getBlocksEndingAtIndex($i$, $OpenB$);
25     $OpenB$.removeAll($EndingBlocks$);
26     $ProcessedB$.addAll($EndingBlocks$);
27 return $ProcessedB$;
```

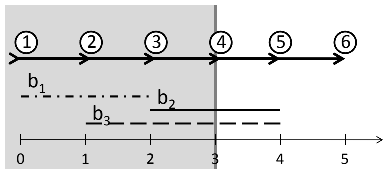
Typically, each block shares comparisons with many other blocks of the input set. However, these pairs of blocks differ in their degree of overlap (i.e., the number of comparisons they share). In fact, *the higher the Jaccard coefficient*[2] *of two blocks, the more comparisons their merge saves*. Thus, to maximize the effect of Block Merging, each block should be merged with the block that has the highest proportion of common entities with it. We call this block **maximum Jaccard block**.

We can now formally define the problem of Block Merging as follows:

Problem 5 (Block Merging). *Given a mined set of blocks B, identify for each block b_i its maximum Jaccard block b_j, and merge these b_i b_j pairs, so as to produce a semantically equivalent set B' with a smaller number of redundant comparisons. We call B' a* ***merged set of blocks****.*

To address this problem, we introduce Algorithm 4. At its core lies the idea that each block should have overlapping spatial representations with its maximum Jaccard block. In other words, there is no point in estimating the Jaccard similarity between two blocks with disjoint spatial representations. Based on this principle, our algorithm works for unilateral blocks as follows: two main lists are maintained, the Open and the Processed ones (lines 4 and 5, respectively). The former contains the blocks that are available for comparison, whereas the latter encompasses the blocks that do not

[2]The Jaccard similarity coefficient $J(A, B)$ of two sets A and B is equal to: $J(A, B) = \frac{|A \cap B|}{|A \cup B|}$.

(a)


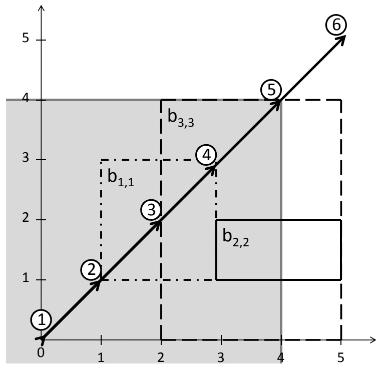
(b)


Figure 2: Illustration of the block merge algorithm.

need to be considered any more. Starting from the beginning of the X-axis, the algorithm traverses the X-axis by one point in every iteration (line 6); blocks, whose spatial representation starts from the current point (line 7), are compared with the blocks in the Open list (line 14). Then, at the completion of the iteration, they are added in it, together with the merged blocks (lines 23 and 8, respectively). Blocks, whose spatial representation ends at the current point, are placed in the Processed list (lines 24-26). The reason is that they do not overlap with any of the subsequently examined blocks. In the case of bilateral blocks, the only difference in the execution of the algorithm is that it traverses both axes simultaneously. The following example illustrates the functionality of this algorithm.

Example 5. *Figure 2(a) is an illustration of the Block Merging algorithm for unilateral blocks that are mapped to the X-axis. The main idea is that the grey area expands by one unit after each step. All blocks lying partly within it are in the OpenB list, while all blocks that lie entirely within it are placed in the ProcessedB list. At the highlighted Step 4, b_2 and b_3 lie in the former list, while b_1 is placed in the latter. The execution of the algorithm is as follows: at Step 1, only b_1 is in OpenB, while, at Step 2, b_3 is also added and compared with b_1. At Step 3, b_1 is removed from OpenB. Then, b_2 is placed in OpenB and compared with all other blocks. The remaining blocks, b_2 and b_3, are placed in ProcessedB at the end of Step 5.*

Figure 2(b) is an illustration of the block merge algorithm for bilateral blocks that are mapped to the XY-space. Again, the grey area expands at each step by one unit, in both dimensions this time. All blocks lying partly within it are placed in the OpenB list, while all blocks that lie entirely inside its borders are moved to the ProcessedB list. In the depicted case (step 5), $b_{1,1}$ lies in the ProcessedBlocks list, while $b_{2,2}$ and $b_{3,3}$ are in the OpenBlocks one.

As mentioned in Definition 5, the input to Block Merging is a mined set of blocks; that is, the input set contains neither identical nor dominated blocks, since they do not contribute non-redundant comparisons. The computational cost of Algorithm 4 is thus significantly reduced, without affecting its output. However, during the execution of Algorithm 4, blocks that belong to one of these categories can be produced, and should be discarded on-the-fly. Indeed, merges that lead to a block identical to another block of the input set are immediately removed (lines 9-10 Algorithm 4). Regarding new relations of dominance, merges can be involved in them only as dominant blocks. Otherwise, the original blocks that produce them would have already been dominated. More formally:

$$b_{m_{i,j}} \preceq b_k \Leftrightarrow (b_i \cup b_j) \preceq b_k \Rightarrow (b_i \preceq b_k) \wedge (b_j \preceq b_k).$$

Thus, only a block of the input set can be dominated by the merge of two other blocks. Apparently, it is not efficient to apply the Block Mining method after each iteration of Block Merging. Dominated blocks are, therefore, removed after the completion of the merging process, by comparing the original, non-merged blocks with the set of merges.

6. EVALUATION

In this section we present a series of experiments that investigate the higher efficiency conveyed by our techniques, when incorporated into existing, redundancy-bearing blocking methods. Our techniques were fully implemented in Java 1.6, and the experiments were performed on a server with Intel Xeon 3.0GHz.

Metrics. Given that our techniques aim at eliminating redundant comparisons, we evaluate them on the basis of an established metric for measuring the efficiency of blocking methods, namely the **Reduction Ratio** (*RR*) [1, 3, 18]. It expresses the reduction in the number of pair-wise comparisons required by a method with respect to the baseline one. Thus, it is defined as follows: $RR = 1 - mc/bc$, where mc stands for the number of comparisons entailed by our technique, and bc expresses the number of comparisons entailed by the baseline (in our case, this is the original blocking method). *RR* takes values in the interval $[0, 1]$ (for $mc \leq bc$), with higher values denoting higher efficiency.

Recall (named *Pair Completeness* in the context of blocking [21]) is not reported, since our techniques do not affect the duplicates identified by a blocking method - their goal is exclusively to detect and avoid the superfluous and repeated comparisons.

Data Sets. To evaluate the impact of our techniques on existing blocking methods, we employ two real-world data sets; one for the Clean-Clean, and one for the Dirty-Dirty and Dirty-Clean cases of ER. They are *DBPedia* and the *BTC*09 data set, respectively, which are described in more detail in the following paragraphs.

*DBPedia Infobox Dataset (**DBPedia**).* This data collection consists of two different versions of the DBPedia Infobox Data Set[3]. They have been collected by extracting all name-value pairs from the infoboxes of the articles in Wikipedia's English version, at specific points in time. Although it may seem simple to resolve two versions of the same data set, this is not the case. More specifically, the older version (*DBPedia*$_1$) is a snapshot of Wikipedia Infoboxes in October 2007, whereas the latest one (*DBPedia*$_2$) dates from October 2009. During the two years that intervene between these two versions, Wikipedia Infoboxes were so heavily modified that there is only a small overlap between their profiles, even for duplicate entities: just 25% of all name-value pairs are shared among the entities common in both versions. As matches, we consider those entities that have exactly the same URL in both versions. The attribute-agnostic blocking method introduced in [21] was applied on this data set to produce the set of *bilateral* blocks we employ in our experiments. The blocks we consider are those resulting from the Block Purging step [21], involving 3.98×10^{10} comparisons and $PC = 99.89\%$. The technical characteristics of *DBPedia* are presented in Table 1.

[3]See `http://wiki.DBPedia.org/Datasets`.

DBPedia	$DBPedia_1$ Entities	1, 190, 734
	$DBPedia_1$ Name-Value Pairs	17, 453, 516
	$DBPedia_2$ Entities	2, 164, 058
	$DBPedia_2$ Name-Value Pairs	36, 653, 387
	Bilateral Blocks	1, 210, 262
	Comparisons	3.98×10^{10}
	Av. Comparisons Per Block	32, 893
	Av. Blocks Per Entity	15.38
BTC09	Entities	1.82×10^8
	Name-Value Pairs	1.15×10^9
	Unilateral Blocks	8.04×10^7
	Comparisons	4.05×10^9
	Av. Comparisons Per Block	50.37
	Av. Blocks Per Entity	2.61

Table 1: Technical characteristics for both data sets.

*Billion Triple Challenge 2009 (**BTC09**).* This data set constitutes a publicly available[4], large snapshot of the Semantic Web, aggregating RDF statements from a variety of sources. In total, it comprises 182 million distinct entities, described by 1.15 billion triples. The ground-truth set of duplicate entities is derived from the *explicit* as well as *implicit* equivalence relationships (i.e., the `owl:sameAs` statements and the Inverse Functional Properties, respectively). $BTC09$ was employed as a test bed for the attribute-agnostic blocking method presented in [22], which is the source of our experimental set of *unilateral* blocks. well. Similar to *DBPedia*, the blocks we employ in our experiments are those stemming from the Block Purging method of [22]. They entail 4.05×10^9 comparisons, exhibiting a PC very close to 90%. A more detailed overview of this data set is presented in Table 1.

6.1 Block Mapping

In this section, we compare Algorithm 1 with the Random Mapping (**RM**) of blocks to the Cartesian space. As mentioned in Section 4.2, the higher the performance of a mapping method, the lower the sum of Formula 1 should be. Table 2 shows the outcome for both algorithms and for both data sets. In each case, we considered 100 iterations with RM in order to get a safe estimation of its performance. The sums in Table 2 is equal to the average value. It is remarkable that standard deviation is equal with $\pm 4.16 \times 10^{16}$ and $\pm 3.97 \times 10^{12}$ for *DBPedia* and $BTC09$, respectively, thus being 4 orders of magnitude lower than the mean value in both cases. This indicates that the performance of RM is relatively stable.

We can see that Algorithm 1 substantially outperforms RM in both cases; for bilateral blocks, it improves RM by 30.34%, whereas for unilateral blocks the enhancement is 25.28%. The reason for the slightly lower improvement in the second case is twofold. First, redundancy is much higher in *DBPedia* than in $BTC09$, as documented in Table 1; in the former data set, an entity is associated with more than 15 blocks, in comparison with the less than 3 blocks for the latter. Thus, mapping an entity to a suboptimal (random) coordinate affects the spatial deviation of more blocks in *DBPedia* than in $BTC09$. Second, suboptimal mappings have a larger impact in the two-dimensional space than in the unidimensional one. Consider for example Figure 1. Assigning entity id_4 to point 5 of the X-axis instead of 4, increases the spatial deviation of b_2 by 1. However, assigning entity $id_{1,4}$ to point 5 of the Y-axis instead of 4 increments the spatial deviation of $b_{3,3}$ by 4. For these

[4]See `http://vmlion25.deri.ie`.

Method	DBPedia Sum	BTC09 Sum
Algorithm 1	0.81×10^{20}	1.33×10^{16}
Random Mapping	1.16×10^{20}	1.78×10^{16}

Table 2: Comparison of the Block Mapping technique with respect to the sum of Formula 1 for both data sets.

Method	DBPedia Comp.	BTC09Comp.
Cartesian Product	5.11×10^{11}	3.23×10^{19}
LSH + Algorithm 3	1.27×10^8	4.52×10^9

Table 3: Performance of the Block Mining algorithm.

	DBPedia		BTC09	
Method	Comp.	RR	Comp.	RR
Input Set	3.98×10^{10}	-	4.01×10^9	-
Comp. Prop.	2.59×10^9	93.49%	3.08×10^9	23.19%

Table 4: Effect of Comparisons Propagation on the efficiency of both data sets.

reasons, the performance of RM is slightly closer to Algorithm 1 for unilateral blocks.

6.2 Block Mining

We now present the performance of our Block Mining algorithm (i.e., LSH and Algorithm 3) with respect to the number of block comparisons it requires in order to examine the blocks inside each bucket. In general, to determine the structure of LSH, two parameters have to be specified: $\mathcal{L}$ and k; the former denotes the total number of hash tables that will be employed, while the latter specifies the number of hash functions that are merged to compose the hash signature for each entity, for a particular hash table. In our implementation, we followed [23] and used *exclusive or* for efficiently merging the k hash functions into the composite signature. In more detail, $\mathcal{L}$ was set to 10, and k to 12, while the probabilities p_1 and p_2 (see Section 5.2.2) were set to 0.9 and 0.1, respectively.

Table 3 shows the performance of our method in comparison with the naive method of examining all possible pairs of blocks. The aim is not to examine the optimal configuration of LSH. Rather, the main conclusion to be drawn from these numbers is that it is a scalable approach, whose performance depends on the level of redundancy of the underlying blocking method: the higher the redundancy, the more similar the blocks are between them, and the larger the corresponding buckets get; this results in more comparisons and lower efficiency. This explains why the performance of our method is 10 orders of magnitude better than the Cartesian Product for $BTC09$, which has low redundancy, while for *DBPedia*, which has high redundancy, it is reduced to 3 orders of magnitude.

6.3 Effect on Blocking Methods

Comparisons Propagation. Table 4 presents the improvement in efficiency conveyed by Comparisons Propagation to both blocking techniques. For *DBPedia*, the number of required comparisons drops by a whole order of magnitude, while for $BTC09$ almost a quarter of all comparisons are saved. Apparently, the reason for the lower improvement in the latter case is the difference in the degree of redundancy: in *DBPedia* it is almost 5 times higher, with each block placed into 15 blocks, on average, in contrast with the less than 3 for $BTC09$. This clearly demonstrates that the higher the redundancy of a blocking method, the higher the enhancement in the efficiency that Comparisons Propagation brings about.

Method	DBPedia Comp.	DBPedia RR	BTC09 Comp.	BTC09 RR
Input Set	3.98×10^{10}	-	4.05×10^9	-
Block Cleaning	3.96×10^{10}	0.05%	3.97×10^9	1.98%
Block Mining	3.88×10^{10}	2.51%	3.80×10^9	6.17%
Block Merging	2.58×10^{10}	35.18%	3.48×10^9	14.08%

Table 5: Effect of each step of the Block Manipulation method on the efficiency of both data sets.

Block Category	DBPedia #Blocks	DBPedia Perc.	BTC09 #Blocks	BTC09 Perc.
Input Set	1,210,262	100.00%	80.39×10^6	100.00%
Identical	199,204	16.46%	1.99×10^6	2.48%
Dominated	495,356	40.93%	4.90×10^6	6.10%
Merged	15,407	1.25%	0.78×10^6	0.98%

Table 6: Categorization of the blocks of both data sets.

Block Manipulation. In this section, we individually analyze the contribution of each step of Block Manipulation to its the overall performance. To this end, Table 5 analytically presents the RR of each method, while Table 6 displays the relative size of all block categories for both data sets. We can see that the portion of comparisons discarded by Block Cleaning is negligible in both cases, although a large part of the input sets (especially in *DBPedia*) are identical blocks. This is because such blocks are typically of very small size; the larger a block is, the lower the likelihood that there is another one in the input set with exactly the same entities. This principle applies to the dominated blocks, as well. Consequently, the contribution of Block Mining is also minor, though higher than that of Block Cleaning. However, as seen in Table 6, both steps are necessary for discarding a large portion of blocks that are superfluous for Block Merging. Thus, they considerably reduce its computation cost. Note also that the higher the level of redundancy, the more blocks are removed by these two steps.

Regarding Block Merging, it is clear that it constitutes the most crucial method of Block Manipulation. As demonstrated in Tables 5 and 6, it is responsible for the most significant reduction in superfluous comparisons, although it involves much less blocks. The reason is that the majority of blocks that are merged are large, containing most of the redundant comparisons. It is worth noting, though, that there is plenty of space for improvement, since its performance deviates significantly from that of Comparisons Propagation. To bridge this gap, we plan to develop more efficient Block Merging algorithms in the future.

7. CONCLUSIONS

In this paper, we presented a series of generic methods for eliminating the superfluous comparisons in a redundancy-bearing blocking method. We introduced Comparisons Propagation that provides the optimal solution to this problem at the cost of quadratic space complexity. In addition, we proposed an alternative solution with less space requirements that manipulates blocks as abstract sets of integers, which are mapped to the Cartesian space. It can identify blocks that are subsets of other ones, as well as blocks that are highly overlapping. By discarding the former and merging the latter, we can save a significant amount of comparisons. In the future, we intend to investigate more effective methods for Block Merging, so that their performance gets closer to Comparisons Propagation. We also plan to integrate techniques for parallelization, namely MapReduce, in order to further enhance the efficiency of our methods.

Acknowledgments. This work is partly funded by the European Commission under the FP7 EU Projects GLOCAL (Contract No. 248984) and LivingKnowledge (Contract No. 231126).

References

[1] M. Bilenko, B. Kamath, and R. J. Mooney. Adaptive blocking: Learning to scale up record linkage. In *ICDM*, 2006.
[2] W. W. Cohen, P. D. Ravikumar, and S. E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *IIWeb*, 2003.
[3] T. de Vries, H. Ke, S. Chawla, and P. Christen. Robust record linkage blocking using suffix arrays. In *CIKM*, 2009.
[4] A. Doan and A. Y. Halevy. Semantic integration research in the database community: A brief survey. *AI Magazine*, 2005.
[5] X. Dong, A. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. In *SIGMOD*, 2005.
[6] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Trans. Knowl. Data Eng.*, 2007.
[7] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, 1999.
[8] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, 2001.
[9] H. Han, C. L. Giles, H. Zha, C. Li, and K. Tsioutsiouliklis. Two supervised learning approaches for name disambiguation in author citations. In *JCDL*, 2004.
[10] H. Han, H. Zha, and C. L. Giles. Name disambiguation in author citations using a k-way spectral clustering method. In *JCDL*, 2005.
[11] M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. In *SIGMOD Conference*, 1995.
[12] D. V. Kalashnikov and S. Mehrotra. Domain-independent data cleaning via analysis of entity-relationship graph. *TODS*, 2006.
[13] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: similarity measures and algorithms. In *SIGMOD*, 2006.
[14] D. Lee, B.-W. On, J. Kang, and S. Park. Effective and scalable solutions for mixed and split citation problems in digital libraries. In *IQIS*, 2005.
[15] C. Li, L. Jin, and S. Mehrotra. Supporting efficient record linkage for large data sets using mapping techniques. *WWW J.*, 9(4), 2006.
[16] J. Madhavan, S. Cohen, X. L. Dong, A. Y. Halevy, S. R. Jeffery, D. Ko, and C. Yu. Web-scale data integration: You can afford to pay as you go. In *CIDR*, 2007.
[17] A. McCallum, K. Nigam, and L. H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *KDD*, pages 169–178, 2000.
[18] M. Michelson and C. A. Knoblock. Learning blocking schemes for record linkage. In *AAAI*, 2006.
[19] B.-W. On, N. Koudas, D. Lee, and D. Srivastava. Group linkage. In *ICDE*, 2007.
[20] B.-W. On, D. Lee, J. Kang, and P. Mitra. Comparative study of name disambiguation problem using a scalable blocking-based framework. In *JCDL*, 2005.
[21] G. Papadakis, E. Ioannou, C. Niederée, and P. Fankhauser. Efficient entity resolution for large heterogeneous information spaces. In *WSDM*, 2011.
[22] G. Papadakis, E. Ioannou, C. Niederée, T. Palpanas, and W. Nejdl. Improving the efficiency of entity resolution for large-scale semantic data. In *Technical Report, available at: http://www.l3s.de/ papadakis/papers/infixBlocking.pdf*, 2011.
[23] O. Papapetrou and L. Chen. Xstreamcluster: an efficient algorithm for streaming xml data clustering. In *DASFAA (to appear)*, 2011.
[24] H. sik Kim and D. Lee. Harra: fast iterative hashed record linkage for large-scale data collections. In *EDBT*, 2010.
[25] Y. Song, J. H. 0002, I. G. Councill, J. Li, and C. L. Giles. Efficient topic-based unsupervised name disambiguation. In *JCDL*, 2007.
[26] S. Tejada, C. A. Knoblock, and S. Minton. Learning domain-independent string transformation weights for high accuracy object identification. In *KDD*, 2002.
[27] P. Treeratpituk and C. L. Giles. Disambiguating authors in academic publications using random forests. In *JCDL*, 2009.
[28] S. E. Whang, D. Menestrina, G. Koutrika, M. Theobald, and H. Garcia-Molina. Entity resolution with iterative blocking. In *SIGMOD Conference*, 2009.
[29] S. Yan, D. Lee, M.-Y. Kan, and C. L. Giles. Adaptive sorted neighborhood methods for efficient record linkage. In *JCDL*, 2007.