BLAST2 : An Efficient Technique for Loose Schema Information Extraction from Heterogeneous Big Data Sources

DOMENICO BENEVENTANO, SONIA BERGAMASCHI, LUCA GAGLIARDELLI, and GIOVANNI SIMONINI, Università degli Studi di Modena e Reggio Emilia

We present *BLAST2*, a novel technique to efficiently extract *loose schema information*, i.e., metadata that can serve as a surrogate of the schema alignment task within the Entity Resolution (ER) process, to identify records that refer to the same real-world entity when integrating multiple, heterogeneous, and voluminous data sources. The *loose schema information* is exploited for reducing the overall complexity of ER, whose naïve solution would imply $O(n^2)$ comparisons, where n is the number of entity representations involved in the process and can be extracted by both structured and unstructured data sources. *BLAST2* is completely unsupervised yet able to achieve almost the same precision and recall of supervised state-of-the-art schema alignment techniques when employed for Entity Resolution tasks, as shown in our experimental evaluation performed on two real-world datasets (composed of 7 and 10 data sources, respectively).

CCS Concepts: • **Information systems** → **Entity resolution**; **Deduplication**;

Additional Key Words and Phrases: Entity resolution, data integration, big data

ACM Reference format:
Domenico Beneventano, Sonia Bergamaschi, Luca Gagliardelli, and Giovanni Simonini. 2020. *BLAST2* : An Efficient Technique for Loose Schema Information Extraction from Heterogeneous Big Data Sources. *J. Data and Information Quality* 12, 4, Article 18 (November 2020), 22 pages.
https://doi.org/10.1145/3394957

1 INTRODUCTION

In recent years, thanks to the growing awareness of the potential value of the data[1] and to the constant decrease of the costs for storing it, companies and organizations started to gather huge amounts of data related to any aspect of their business, even when not knowing in advance what that data might be useful for. For example, for a manufacturing company, it is common to collect along structured data about products and customers (just to name a few): semi-structured data about machinery and sales agents activities coming from web services and unstructured data about product reviews collected from the web.

[1] https://www.economist.com/leaders/2017/05/06/the-worlds-most-valuable-resource-is-no-longer-oil-but-data.

Authors' address: D. Beneventano, S. Bergamaschi, L. Gagliardelli, and G. Simonini, Università degli Studi di Modena e Reggio Emilia, Modena, Italy; emails: {domenico.beneventano, sonia.bergamaschi, luca.gagliardelli, giovanni.simonini}@unimore.it.

1936-1955/2020/11-ART18 $15.00
https://doi.org/10.1145/3394957

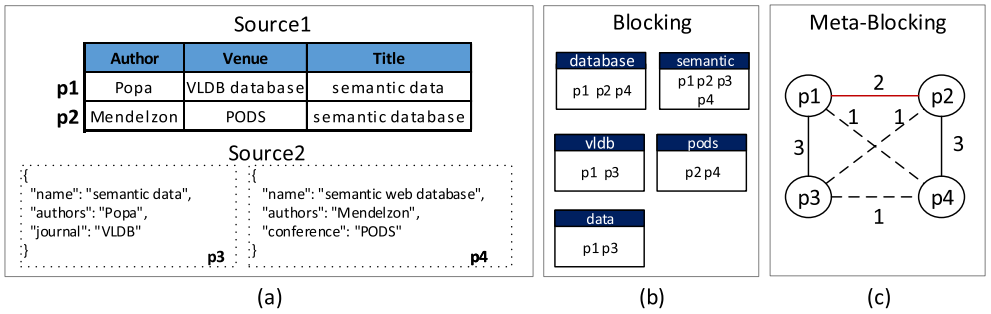

Fig. 1. Schema-agnostic (meta-)blocking process.

One of the main problems with these data is heterogeneity: Data scientists and practitioners deal with data integration on a daily basis to merge datasets and unleash the true value of the data. Being able to identify different representations (*profiles*) that pertain to the same real-world entity in different datasets is a crucial task (known as Entity Resolution or Duplicate Detection) for data integration, as well as for data science.

Another main problem arises due to the emerging increasing size of data sources, i.e., Big Data, so that the Entity Resolution (ER) task has to take into account scalability and computational costs. In fact, the naïve solution of ER consisting of comparing all pairs of profiles is impracticable with large datasets, and thus *blocking* techniques are employed to group similar records and limiting the comparison only among the profiles contained in the same block.

When working with real-world data sources, to define *blocking key* (i.e., the blocking strategy) yielding high recall and precision is a difficult task [6]. In particular, with heterogeneous data, *schema-aware* techniques have two main issues: (i) schema alignment, hardly achievable with a high heterogeneity of the data; (ii) labeled data to train classification algorithms, or human intervention to select which attributes to combine. To overcome these problems, the *schema-agnostic* approach was introduced [17]: Each profile is treated as a *bag of words* and schema-information is ignored. For instance, *Schema-Agnostic Token Blocking* considers as blocking key each token that appear in profiles, regardless of the attribute in which it appears, i.e., each pair of profiles sharing at least one token is considered as a candidate match (Figure 1(a) and (b)).

However, *schema-agnostic* methods typically produce a very low precision: By employing each token as a blocking key, the likelihood that many tokens yield superfluous comparisons is high [16]. So, to alleviate this problem, they have been often coupled with *meta-blocking* [10, 17, 18, 21]. *Meta-blocking* aims to remove from a blocking collection the least-promising comparisons. This is achieved by adopting the following model: Profiles and comparisons are respectively represented as nodes and edges of a graph (i.e., a node is linked to another node if the corresponding profiles co-occur in at least one block). Then, the edges are weighted on the basis of the co-occurence of its adjacent profiles and for each profile a threshold is computed. Finally, the graph is pruned removing the edges that have a weight lower than the threshold. In particular, Figure 1(c) shows the effectiveness of meta-blocking: Each edge is weighted counting the co-occurring blocks of its adjacent profiles and is retained if its weight is above the average. The dashed lines are the removed comparisons, whereas the red ones are superfluous (i.e., comparison of non-matching profiles).

In Reference [21], we proposed *BLAST*, which introduces the notion of *loose schema information*, i.e., metadata extracted from the data in terms of (i) *attribute partitioning* and (ii) *aggregate entropy* (Figure 2(a)). The idea beyond *attribute partitioning* is that the more values two attributes share, the

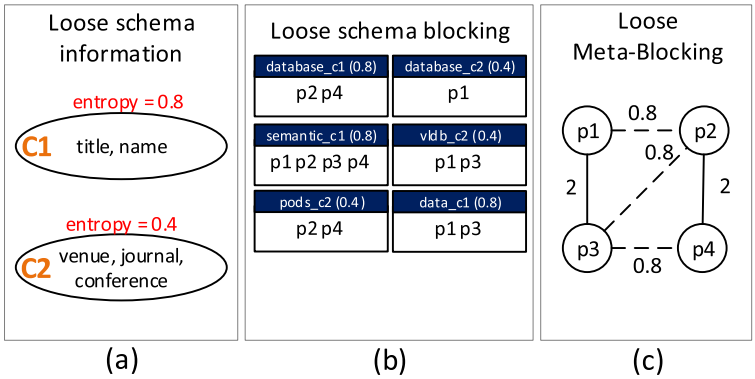

Fig. 2. Meta-blocking with loose schema information.

more they are similar, thus similar attributes are put together in the same partition. Then, the *meta-blocking* takes into account the generated attributes' partitions: The blocking key is composed by tokens concatenated to partition IDs; in this way, for example, the token "Database" (Figure 2(b)) is split into two tokens, disambiguating "Database" as the name of a paper ("Database_C1"), and "Database" as the name of a conference. In other words, on the basis of the loose schema information the blocking key "Database" is disambiguated, then the profiles p1 and p4 share one less block: As a consequence, the edge e1-4 decreases its weight from 3 to 2.

Aggregate entropy computes the entropy of each cluster and gives more importance to the profiles that co-occurs in blocks generated from clusters with high entropy. The idea is that finding equalities inside a cluster with a high variability of the values (i.e., high entropy) has more value that finding them in a cluster with low variability (i.e., low entropy). The *aggregate entropy* is used to improve the edges weights: Each edge of the *meta-blocking* graph is re-weighted according to the entropy associated to the block that generates it (i.e., the entropy of the partition from which the blocking key belongs), as shown in Figure 2(c). This affects the meta-blocking by helping to remove more superfluous comparisons than the ones removed by schema-agnostic blocking (the three retained red edges of Figure 1(c) are now removed).

At the end of the pruning step, the *meta-blocking* produces the *candidate pairs*, i.e., pairs of profiles related to the same entity. Then, these pairs have to be resolved, i.e., it is necessary to decide whether a pair is a true match or not; this task is called *entity matching*. Several techniques can be applied to perform this task, e.g., resolution functions, classifiers, crowdsourcing, and so on. Finally, the retained matching pairs are clustered (*entity clustering*) to group together all the profiles associated to the same entity.

1.1 Contributions

As demonstrated in our previous work [21], loose schema information can be exploited for significantly improving blocking and meta-blocking. Yet, we notice that additional metadata can be extracted efficiently and exploited for further improving the whole ER process. The intuition is to try to detect subsets of the data sources that most likely will contain duplicate profiles (we call this metadata *loose duplicate information*) and try to extract the loose schema information only from that portion of data. This is because (i) it permits to ease the extraction thanks to a significantly smaller amount of data considered; and (ii) the extracted information tends to be less noisy, since it was derived from a portion of the data that is more dense in terms of duplicated profiles. To show this second aspect, we add some profiles to Source 2 (see Figure 3). These new data

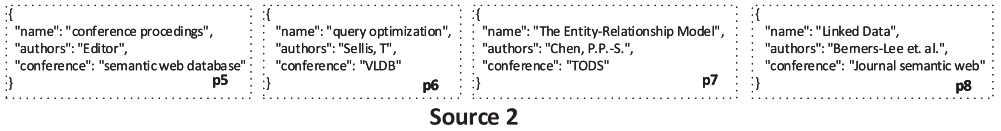

Fig. 3. New profiles added to Source 2 in Figure 1.

without duplicate profiles deteriorate *BLAST*'s performance; in fact, since the attribute partitioning is based on *shared values*, the attribute Conference shares many values with Title and Name, which will then be placed in the same cluster, so obtaining $C1 = \{\text{Title, Name, Conference}\}$ and $C2 = \{\text{Venue, Journal}\}$. This new attribute partitioning nullifies the effects of the *loose schema information* on the blocking key "Database," that is, it is no longer possible to disambiguate between "Database" in the Title of a paper and "Database" in the name of a Conference.

In other words, extracting *loose schema information* from a subset of entity profiles that are likely to match (i.e., similar).

Moreover, by limiting the extraction to those profiles, there is the consequent advantage of processing a significantly smaller amount of data, which translates to a significantly faster execution time, as demonstrated in our experiments (see Section 4). We also propose an efficient method to extract the loose duplicate information alongside with the loose schema information, making out a new method, *BLAST*2 significantly more efficient than *BLAST*.

In detail, in this article we make the following contributions:

- We present an algorithm able to extract loose schema information and loose duplicate information, simultaneously.
- We show how to exploit the loose duplicate information extracted by portions of the datasets with a new technique named *BLAST*2, which speeds up (and in some cases also improve precision) of *BLAST*.
- We experimentally evaluate our novel technique on real-wold multi-source datasets—seven datasets about movies and 10 datasets about books, gathered from openly available sources of the Web—showing how it outperforms the current state-of-the-art (meta-)blocking techniques.

The proposed method is implemented in a new version of *SparkER*, a distributed entity resolution tool, composed by different modules designed to be parallelizable on Apache Spark [9]. More precisely, *BLAST*2 is implemented in a blocker module that takes the input profiles and performs the blocking phase, providing as output the candidate pairs. Other two modules are provided to perform the complete ER process: an entity matcher module that takes the candidate pairs generated by the blocker and labels them as match or no-match; an entity clusterer module that takes the matched pairs and groups them into clusters that represents the same entity.

We are currently integrating the *SparkER* tool in the MOMIS Data Integration System, which is [1, 2] based on a classical wrapper/mediator architecture, where data are stored only at the level of local data sources, while a *Global Virtual Schema*, gives an integrated virtual view of the data sources. In the present article, MOMIS is used as a *schema matching* tool to obtain a Gold Standard for the datasets used for the experiments (see Section 4). MOMIS has among its main features the fast and semi-automatic development of data integration projects; the core of the system is based on a process that discovers semantic similarities between local schemes and automatically generates a Global Virtual Schema and its mappings with local schemes. Such mappings represent an attribute partitioning of local attributes and, then, we can obtain the Gold Standard, for the considered datasets.

The article is organized as follows. The next section contains preliminaries and related work. In Section 3, we present *BLAST2*, which significantly improves *BLAST* as it works on a significant smaller amount of data (portions of the datasets without loosing precision and recall). In Section 4, we describe the setup and the datasets used for the experiments. Finally, Section 5 presents our conclusions.

2 PRELIMINARIES AND RELATED WORK

This section describes the fundamental concepts and notation employed in this article following the definitions of Reference [21], while presenting the related work on which our work is built.

2.1 Blocking for Entity Resolution

An entity *profile* is a tuple consisting of a unique identifier and a set of *name-value* pairs $\langle a, v \rangle$. $A_{\mathcal{P}}$ is the set of possible attributes a associated to a profile collection $\mathcal{P}$. A *profile collection* $\mathcal{P}$ is a set of profiles. Two profiles $p_i, p_j \in \mathcal{P}$ are *matching* ($p_i \approx p_j$) if they refer to the same real-world object; *Entity Resolution* (ER) is the task of identifying those matches given $\mathcal{P}$.

The naïve solution to ER implies $|\mathcal{P}_1| \cdot |\mathcal{P}_2|$ comparisons, where $|\mathcal{P}_i|$ is the cardinality of a profile collection $\mathcal{P}_i$. *Blocking* aims to reduce this complexity by indexing similar profiles into *blocks* according to a *blocking key* (i.e., the indexing criterion), restricting the actual comparisons of profiles to those appearing in the same block.

We call *block collection* a set of blocks $\mathcal{B}$, its *aggregate cardinality* is the cardinality of the set of comparison $C^{\mathcal{B}}$ entailed by the blocking collection $\mathcal{B}$, i.e., $|C^{\mathcal{B}}| = \sum_{b_i \in \mathcal{B}} \|b_i\|$, where $\|b_i\|$ is the number of comparisons derived from the block b_i. Following the best practices for blocking evaluation [14, 17], our model employs an *Entity Resolution Algorithm* to determine if two profiles are actually duplicates (i.e., refer to the same real-world entity). As a matter of fact, *BLAST* is independent of such an algorithm—just as the other state-of-the-art blocking techniques [8, 17].

Recall and *Precision* are employed in this article to evaluate the quality of a block collection $\mathcal{B}$. The recall measures the fraction of duplicates that are indexed in the blocks, i.e., how many duplicates can be identified; while the precision measures the fraction of useful comparisons, i.e., how many comparisons correspond to true duplicates.

Formally, for a block collection $\mathcal{B}$:

$$recall = \frac{|C^{\mathcal{B}} \bigcap \mathcal{D}^{\mathcal{P}}|}{|\mathcal{D}^{\mathcal{P}}|}; \qquad precision = \frac{|C^{\mathcal{B}} \bigcap \mathcal{D}^{\mathcal{P}}|}{|C^{\mathcal{B}}|};$$

where $C^{\mathcal{B}}$ is the set of comparison pairs entailed by the block collection $\mathcal{B}$ and $\mathcal{D}^{\mathcal{P}}$ is the set of all the duplicates in $\mathcal{P}$.

We call *redundant* comparisons the comparison of profiles that are entailed in the blocking collection more than once (i.e., when two profiles are together in more than one block); and we call *superfluous* comparisons the comparison of non-matching profiles ($p_i \not\approx p_j$). For example, in Figure 1(c), the comparison between p_1 and p_3 is redundant, since such profiles are in the "semantic," "vldb," and "data" blocks; the comparison between p_1 and p_2 is superfluous, since they are non-matching profiles.

Attribute-match induction [2] approaches aim to enhance schema-agnostic blocking by avoiding (some of) the superfluous comparisons. *Meta-blocking* is an approach aiming to reduce both superfluous and redundant comparisons restructuring a given block collection. In the following, both attribute-match induction and meta-blocking are formally defined.

[2]We refer as *attribute-match induction* for identifying all the approaches that group similar attributes, while we refer to the specific technique proposed in Reference [14] with *Attribute Clustering*.

2.2 Attribute-match Induction

The goal of attribute-match induction is to group together *similar* attributes of two profile collections directly from the distribution of their values, without leveraging on the semantics of their names. This information about how the attribute can be grouped is then exploited to support a schema-agnostic blocking technique, i.e., to disambiguate blocking keys in accordance to the attribute group from which they are derived (e.g., tokens "database" in Figure 3(b)).

Problem 1 (Attribute-match Induction). *Given two collections of profiles* $\mathcal{P}_1, \mathcal{P}_2$, attribute-match induction *is to identify pairs* $\{\langle a_i, a_j\rangle \mid a_i \in A_{\mathcal{P}_1}, a_j \in A_{\mathcal{P}_2}\}$ *of similar attributes on the basis of a similarity measure and to exploit those pairs to partition the attribute name space* $(A_{\mathcal{P}_1} \times A_{\mathcal{P}_2})$ *in non-overlapping clusters, i.e., to yield the* attributes partitioning.

This task is fundamentally different from the traditional schema-matching: The latter aims to detect exact correspondences, hierarchies, and containment among attributes [20].

An attribute-match induction task is defined by four components, which are formally presented later in this section: (i) the *value transformation function*, (ii) the *attribute representation model*, (iii) the *similarity measure* to match attributes, and (iv) the *clustering algorithm*.

(1) **The value transformation function.** Given two profile collections $\mathcal{P}_1$ and $\mathcal{P}_2$, each attribute is represented as a tuple $\langle a_j, \tau(V_{a_j})\rangle$, where $a_j \in A_{\mathcal{P}_i}$ is an attribute name; V_{a_j} is the set of values that an attribute a_j can assume in $\mathcal{P}_i$; and τ is a *value transformation function* returning the set of *transformed* values $\{\tau(v) : v \in V_{a_j}\}$. The function τ typically is the concatenation of text transformation functions (e.g., *tokenization*, *stop-words* removal, *lemmatization*). Given a τ transformation function, the set of possible values in the profile collections is $T_A = T_{a_{\mathcal{P}_1}} \bigcap T_{a_{\mathcal{P}_2}}$, where $T_{a_{\mathcal{P}}} = \bigcup_{a_i \in A_{\mathcal{P}}} \tau(V_{a_i})$.

(2) **The attribute representation model.** An attribute a_i is represented as a vector $\overrightarrow{\mathcal{T}_i}$ (named the *profile* of a_i), where each element $v_{in} \in \overrightarrow{\mathcal{T}_i}$ is associated to an element $t_n \in T_A$. If $t_n \notin \tau(V_{a_i})$, then v_{in} is equal to zero. While, if $t_n \in \tau(V_{a_i})$, then v_{in} assumes a value computed employing a weighting function, such as [14]: *TF-IDF*(t_n) or the *binary-presence* of the element t_n in $\tau(V_{a_i})$ (i.e., $v_{in} = 1$ if $t_n \in \tau(V_{a_i})$, 0 otherwise). For example, say that the value transformation function τ is the *tokenization* function and that the function to weight the vector elements is the *binary-presence*. Then, the attributes are represented as a matrix: Each row corresponds to an attribute, each column corresponds to a possible tokens appearing in the profile collections, and each element v_{in} is either 1 (if the token t_n appear in the attribute a_i) or 0 (otherwise).

(3) **The similarity measure.** A *similarity measure* is used to compare the profiles $\overrightarrow{\mathcal{T}_j}$ and $\overrightarrow{\mathcal{T}_k}$ of each attribute pair $(a_j, a_k) \in (A_{\mathcal{P}_1} \times A_{\mathcal{P}_2})$. Typical similarity measure, such as Jaccard and Cosine, can be used; the only requirement is that the similarity measure has to be compatible with model used for attribute representation; for example, the *Jaccard* similarity cannot be used with the *TF-IDF* weighting.

(4) **The clustering algorithm.** The input of a clustering algorithm are the attribute pairs with the computed similarities of their profiles, which are exploited to perform the partitioning of the attribute names. (See Section 3.1 for more details.) The output of a clustering algorithm is called *attributes partitioning*; notice that we use non-overlapping partitioning.

2.3 Meta-blocking

By employing redundant blocking techniques—i.e., extracting multiple blocking keys from each profile—the chance of missing comparisons that correspond to matches decreases, but the downside is that many superfluous/redundant comparisons typically are introduced (e.g., two profiles

may appear together in more than one block yielding a redundant comparison). The goal of *meta-blocking* [17] is to restructure a block collection, built with a redundant blocking technique, by removing superfluous/redundant comparisons: If the number of those comparisons decreases, then the recall is not affected, while the precision increases. This *comparison pruning* is done in *meta-blocking* by exploiting the intuition that the more blocks two profiles appear in together, the more likely they will be considered duplicates by the Entity Resolution algorithm. In the following a more formal presentation of *meta-blocking* is given.

Problem 2 (Meta-blocking). *Given a block collection* $\mathcal{B}$, meta-blocking *is the task to restructure the blocks in a new block collection* $\mathcal{B}'$ *such that* $precision(\mathcal{B}') \gg precision(\mathcal{B})$ *and* $recall(\mathcal{B}') \simeq recall(\mathcal{B})$.

In *graph-based meta-blocking* (or simply *meta-blocking* from now on), a block collection $\mathcal{B}$ is represented by a weighted graph $\mathcal{G}_{\mathcal{B}}\{V_{\mathcal{B}}, E_{\mathcal{B}}, \mathcal{W}_{\mathcal{B}}\}$ called ***blocking graph***. V is the set of nodes representing all $p_i \in \mathcal{P}$. An edge between two entity profiles exists if they appear in at least one block together: $E = \{e_{ij} : \exists p_i, p_j \in \mathcal{P} \mid |\mathcal{B}_{ij}| > 0\}$ is the set of edges; $\mathcal{B}_{ij} = \mathcal{B}_i \cap \mathcal{B}_j$, where $\mathcal{B}_i$ and $\mathcal{B}_j$ are the set of blocks containing p_i and p_j, respectively. $\mathcal{W}_{\mathcal{B}}$ is the set of weights associated to the edges. Meta-blocking methods weight the edges to capture the *matching likelihood* of the profiles that they connect. For instance, ***block co-occurrence frequency*** (a.k.a. CBS) [15, 19] assigns to the edge between two profiles p_u and p_v a weight equal to the number of blocks they shares, i.e., $w_{uv}^{CBS} = |\mathcal{B}_u| \cap |\mathcal{B}_v|$. Then, to keep only more promising edges, we can apply suitable edge-pruning strategies. In this way, after this pruning step, each connected pair of nodes forms a new block of the restructured blocking collection.

Meta-blocking can operate by keeping all the candidate comparisons that are weighted above a certain threshold or in a top-k fashion. We call the first case *Weighted Pruning*, while *Cardinality Pruning* the latter. The weight threshold, or the k for the top-k approach, can be defined at the local level (i.e., for each profile/node in the graph), or at the global level (i.e., for all the edges in the graph). Hence, the combination of those strategies yields the following *pruning strategies*: (i) *Weight Edge Pruning*, where edges with a weight lower the given threshold are pruned; (ii) *Cardinality Edge Pruning*, where edge are sorted in descending order with respect to their weights, and then only the first K are kept; (iii) *Weight Node Pruning* considers in turn each node p_i and its adjacent edges, and edges that are lower than the given threshold are pruned; and (iv) *Cardinality Node Pruning* similarly to the previous one is node centric, but a cardinality threshold k_i is used instead of a weight threshold.

2.4 Locality Sensitive Hashing

In this section, we introduce the basics of *Locality Sensitive Hashing*, an approximate technique to discover similar item sets. In particular, we focus on *MinHash*, for Jaccard Similarity [12].

Locality Sensitive Hashing (LSH) is a family of hash functions that aims to reduce the dimensionality of a high-dimensional space, while preserving the similarity distances. In the context of this article, we are interested in set-based similarities for identifying attributes belonging to different data source that has similar set of values; hence we focus on Jaccard similarity and *MinHash*-based LSH [12].

Algorithm 1 shows how MinHash signatures are computed. The MinHash computation algorithm takes as input a set of elements, in the following we consider sets of attributes, for showing how LSH can be employed for attribute-match induction by representing each attribute as a set of values, i.e., a column of a dataset is represented as a vector whose elements are the values appearing in the cells of the column—optionally transformed with string transformation functions, e.g., *lowercase*, and so on. Then, a set of m hash functions are applied to each cell of the columns,

ALGORITHM 1: MinHash Computation

Require: A set of elements S, which are either the columns or the rows of a dataset; the size of S is n
Require: A set H of m hash functions: $\{\phi_1, \phi_2, \ldots, \phi_m\}$
Ensure: The MinHash matrix M // $M[i]$ is the signature of the $i-th$ element (i.e., column/row depending on the input)

```
M = new Matrix.shape(n, m, +∞) // A n × m matrix whose elements are initialized to +∞, e.g., by using INT_MAX
for each e_i ∈ S do
    for each cell ∈ e_i do
        for each φ_j ∈ H do
            if φ_j(cell) < M[i][j] then
                M[i][j] = φ_j(cell)
return M
```

ALGORITHM 2: Banding

Require: A $n \times m$ MinHash matrix M
Require: t, a similarity threshold
Ensure: A set C of index pairs of element that have a similarity approximately greater than t.

```
Buckets = Map // e.g., a Dict in Python
r ← estimateNumRowsPerBand(m, t) // Function that estimate the number of rows for band
for each i ∈ range(0, n) do
    counter = 0
    signature = ""
    for each j ∈ range(0, m) do
        signature = concat(signature, M[i][j])
        counter = counter + 1
        if counter == r then
            B = Buckets.getOrInitialize(signature, {})
            B.add(i)
            Buckets.put(signature, B)
            r=0
            signature = ""
for each B ∈ Buckets do
    for each (i,j) ∈ B × B do // For each possible pair of index in the current bucket
        if i<j then
            C.add((i,j))
        if i>j then
            C.add((j,i))
return C
```

and for each function only the minimum value for each column is kept. This value is known as *MinHash* and the n MinHashes of each column are called the *MinHash signature* of the column. Interestingly, the probability of yielding the same MinHash values for two columns, is equal to their Jaccard similarity; thus, MinHash preserves the similarity transforming the matrix, with the advantage of reducing the dimension of the vectors representing the attributes.

Yet, even for small m (e.g., 64), computing the similarity of all possible MinHash signature pairs may be computationally expensive, due to the quadratic complexity of the all-pairs comparison; thus, the signatures are partitioned in *bands* and only signatures that are identical in at least one band are considered—this step is knwon as *banding* and described in Algorithm 2. Overall, if m MinHash values and b bands are employed for the signature generation and banding, respectively, then the probability of two attributes being identical in at least one band is $1 - (1 - s^r)^b$, where

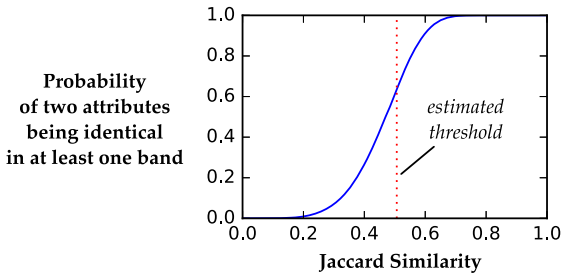

Fig. 4. The depicted curve represents the probability of two attributes to be considered "similar" (y-axis) in function of their actual similarity (x-axis), when LSH is employed (with the parameters $r = 5$ and $b = 30$).

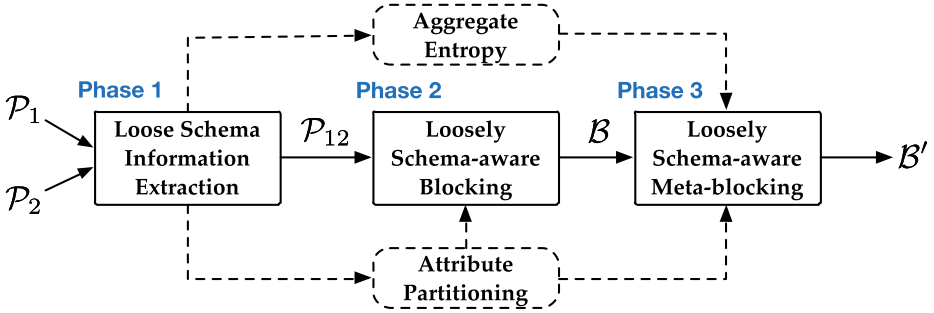

Fig. 5. *BLAST* logical overview.

$r = m/b$ is the number of rows per band. This function has a characteristic S-curve form, and its inflection point represents the threshold of the similarity. The threshold can be approximated to $(1/b)^{1/r}$. For instance, choosing $b = 30$ and $r = 5$, the attribute pairs that have a Jaccard similarity greater than ~0.5 are considered for the attribute-match induction (Figure 4).

3 BLOCKING WITH LOOSE SCHEMA/DUPLICATE INFORMATION

In this section, we introduce the novelties of *BLAST2* w.r.t. *BLAST*. In the first part (Section 3.1), we include a complete description of *BLAST* [22]. Then, in the second part (Section 3.2) we describe *BLAST2* whose core functionality is equivalent to *BLAST*, i.e., both exploit the loose schema information in the same way for blocking and meta-blocking, but it introduces a novel preprocessing step that permits to extract much efficiently the loose schema information.

3.1 BLAST

BLAST is an efficient method to automatically extract loose schema information, which is then used for both blocking and meta-blocking. This holistic combination significantly helps in producing high-quality candidate pairs for ER, compared to other existing meta-blocking techniques, which operates only in completely schema-agnostic setting [8, 15, 17].

For the sake of the presentation, in this section we consider the case of detecting duplicates from two different data sources ($\mathcal{P}_1$ and $\mathcal{P}_2$ in in Figure 5), but our method works in the same exact way also when multiple data source are considered—as a matter of fact, in our experimental evaluation (Section 4) multi-source datasets are employed. The overall workflow of *BLAST* is depicted in Figure 5: the loose schema information is extracted from the data sources (phase 1) and then it is exploited for building (phase 2) and restructuring (phase 3) a blocking collection. In the following, these three phases are described in more details.

Phase 1: The loose schema information is extracted; it is composed of the *attributes partitioning* and the *aggregate-entropy*. The first consists of clusters of attributes (clustered according to their values' similarity); the second measures how *informative* is each of these clusters; together we call these peaces of information: *loose schema information*. For extracting the attributes partitioning, *BLAST* exploits an LSH-based algorithm: minhash signatures are employed to quickly approximate the similarities of the attributes and to build a similarity graph of attributes, which is given as input to a graph-partitioning algorithm (the details of the algorithm can be found in Reference [21]). *BLAST* also extracts the (Shannon) entropy of each cluster of attributes (which is the average of the entropies of its attributes). Basically, the entropy is a measure of how informative is a set of attributes: intuitively, if all the values in the partition are the same, then the entropy is equal to zero, and it should not be used indeed for generating blocking keys, since all the profiles would be indexed in just one unique block. However, if the entropy is high, then the profiles share values from that attribute partition that are less frequent, and hence the derived blocking keys will be more likely useful.

Phase 2: A schema-agnostic blocking technique is applied on each attribute partition obtained in the previous phase. For instance, in our experiments (see Section 4) Token Blocking [16] is employed: Each token is a blocking key, regardless to the attribute of the partition in which it appears in. We call the resulting method Loose-Schema Blocking.

Phase 3: The blocking strategy described in the previous phase allows us to achieve a high level of recall, but it tends to generate a lot of superfluous comparisons. For this reason, a meta-blocking step is performed to generate the final candidate pairs. In particular, *BLAST* leverages on the *entropy* extracted in Phase 1 to weight all the candidate pairs generated in Phase 2. The basic idea is to build a graph (the *Blocking Graph*), where each edge corresponds to a set of blocking keys, and each blocking key is associated to an attribute. Then, the edges are weighted accordingly to their entropy.

For example, consider two independent datasets with people information. Generally, the attribute *birth date* is less informative than the attribute *surname*. This is because the number of distinct birth dates is typically lower than that of the surnames—and the entropy of the former is lower than the entropy of the latter. Thus, *BLAST* assigns a higher weight to edges that represents blocking keys derived from the surnames than those derived from the birth dates. In particular, *BLAST* weights the edges computing the (Pearson) chi-square coefficient—the idea is to measure the significance of the co-occurrence of two profiles in a block—multiplied by the aggregate entropy associated to the blocking keys corresponding to that edge. Finally, *BLAST* applies a local pruning by computing a threshold for each node in the graph (equal to half of the maximum weight of the adjacent edges[3]).

3.2 BLAST2

Here, we present *BLAST2*, which consists in a novel preprocessing step that allows to extract the loose schema information in a significantly more efficient way compare to *BLAST* and even improve the quality of the results in some cases. This is achieved by identifying portions of the datasets that are likely to be dense in duplicates (i.e., subsets of the profiles that match to each other) and extract the loose schema information only from there. The benefit is twofold:

[3]This is a heuristic that has been shown to work well in practice [21].

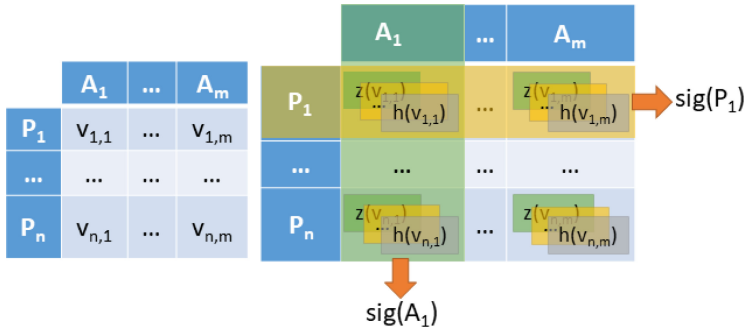

Fig. 6. A dataset matrix (rows are profiles, columns are attributes) used for the extraction of minhash signatures.

(1) by considering less data the loose schema information extraction is faster;
(2) by considering only profiles that are likely to be duplicates, the attribute partitioning tends to be less noisy since less false positive duplicates are considered.

Remember that the attribute partitioning information that we want to mine is employed as a surrogate of the schema alignment for the ER process. Intuitively, if *BLAST* takes as input two sets of profiles that are perfectly duplicated with identical schemas (see in Phase 1 in Section 3.1), then its output would be a perfect schema alignment; hence, in a real-world case, where the datasets are not identical, profiles that have no duplicates are superfluous to this phase (i.e., they represent noise) and can be removed. We call the information about the portions of the datasets that are likely to be dense duplicates *loose duplicate information*.

Interestingly, LSH can be employed for identifying duplicated profiles as well as for determining the attribute partitioning: In the former case, the profiles (i.e., the rows of the datasets) are indexed, gathering together likely-to-match profiles; while, in the latter case, the attributes (i.e., the columns of the datasets) are considered as *document* and indexed though LSH for bucketing together similar attributes. Hence, in *BLAST2* we propose to apply LSH to extract the loose duplicate information to narrow down the search space for the LSH employed for identifying the attribute partitioning. Yet, what is saved by restricting the search space of the attribute partitioning step may not pay off the computational costs. Algorithm 1 shows how MinHash signatures are computed by the traditional LSH [12]—the input can be either the set of rows (first case mentioned above) or the set of columns (second case mentioned above). The output is a matrix of MinHash values—it can also be viewed as an array of MinHash signatures—that has to be processed with Algorithm 2 to yield actual pair of similar rows (or columns). If we want to identify similar row and similar columns, then we need to execute Algorithm 1 and Algorithm 2 twice: one for the rows and one for the columns. Yet, much computation would be redundant, since the same cells are hashed in both the executions—they are just combined differently to generate the MinHash signatures.

In *BLAST2*, we present Algorithm 3 to compute the MinHash signatures of the profiles and the attributes (rows and columns of the datasets, respectively) in a single pass and not as two separated tasks. The main advantage is that Algorithm 3 avoids recomputing the hash functions employed for computing the MinHash signatures of columns and rows. To help in understanding the process of simultaneously computing row and column signatures, let us consider Figure 6, where a dataset is represented as a matrix of profiles and attributes and each cell represents a value of an attribute of a profile. When the MinHash signature of row p_1 is computed, n hash functions are applied to each cell of that row; for instance, $h(v_{11})$ represents the application of the hash function h to the token v_{11}—first row and first column of the matrix. When the MinHash signature of the column

A_1 is computed, the same m hash functions are applied to the rows, for instance $h(v_{11})$. Hence, the opportunity of avoiding the recomputation of all the hash functions. In this way, *BLAST2* is able to compute both row and column MinHashes in one single pass: The time complexity is equivalent to LSH applied on columns (or rows) singularly, and the space complexity is $O(|A| + |P|)$, where $|A|$ is the size of the attribute set and $|P|$ is the size of the profile set—i.e., the space needed to store the signatures of attributes and profiles.

Thus, in conclusion, the output of Algorithm 3 is the *loose schema information*, which is employed for loosely schema-aware meta-blocking (as in *BLAST*), but they way it is extracted is much more efficient compared and its quality is better compared to *BLAST*.

ALGORITHM 3: Combined column-row LSH (Blast2)

Require: A dataset of elements S
Require: A set $\mathcal{H}$ of m hash functions: $\{\phi_1, \phi_2, \ldots, \phi_m\}$
Require: t_{row}, t_{col}, the similarity threshold for columns and rows
Ensure: The set C of column pairs that represent the surrogate schema mapping

```
M = new Matrix.shape(n, m, +∞) // A n × m matrix whose elements are initialized to +∞, e.g., by using INT_MAX
HashCash ← Map // e.g., a Dict in Python to store already computed hashes for a given element
for each Row_i ∈ S do
    for each cell ∈ Row_i do
        if HashCash.get(φ_1(cells)==[] then //if empty
            hashes ← new Array(m) //an empty array of size m (i.e., the number of hash functions)
            for each φ_j ∈ H do
                hashes[j] ← φ_j(cell)
        else
            hashes = HashCash.get(φ_1(cell))
        for each h_j ∈ hashes do
            if M[i][j] < h_j then M[i][j] = h_j
r ← estimateNumRowsPerBand(m, t_row) //Function that estimate the number of rows for band
SimilarRows ← banding(M, r) // Set of rows that have similar rows in the dataset
S' ← filter(S,SimilarRows) // keep only the n' rows that appear in the SimilarRows set
M' = new Matrix.shape(n', m, +∞) // A n' × m matrix whose elements are initialized to +∞, e.g., by using INT_MAX
for each Col_i ∈ S do
    for each cell ∈ Row_i do HashCash.get(φ_1(cell))
        for each h_j ∈ hashes do
            if M'[i][j] < h_j then M'[i][j] = h_j
r ← estimateNumRowsPerBand(m, t_col) // Function that estimate the number of rows for band
C ← banding(M', r) return C
```

4 EVALUATION

In this section, we describe the setup and the datasets used for the experiments that we have conducted. The goal of our experiments is to answer these questions:

(1) What is the performance of *BLAST2* in terms of precision, recall, and execution time compared to `schema-agnostic Token Blocking`, *BLAST*, and *BLAST* modified to use the Gold Standard (i.e., using the Gold Standard as attribute partitioning)?
(2) How does the LSH threshold affect *BLAST2* performance? (Section 4.2)

Setup. All the experiments had been performed on a machine with Intel Xeon E3-12xxv2 3.00 GHz (16 cores) and 100 GB of RAM, running Ubuntu 18.04. All the code is in Scala 2.11.8, and we employ Apache Spark 2.1.0.

Table 1. Dataset Characteristics: Number of Entity Profiles, Number of *Attribute Names*, and the Number of Duplicates in the Ground Truth

Movies small

	#profiles	*#attributes*
Amazon	5.2k	6
IMDB1	6.4k	12
IMDB2	2.9k	10
IMDB3	6.9k	6
RogerErbert	3.5k	8
Rotten1	7.3k	16
Rotten2	3.0k	10

ground-truth

	#duplicates
Books	1.2k
Movies small	21.1k
Movies big	33.1k

Books

	#profiles	*#attributes*
Amazon1	3.5k	11
Amazon2	8.9k	21
Amazon3	3.0k	8
BarnesNoble1	3.5k	11
BarnesNoble2	3.7k	16
BarnesNoble3	3.0k	8
BarnesNoble4	9.9k	13
BarnesNoble5	3.0k	9
GoodReads	3.9k	16
Half	3.1k	10

Movies big

	#profiles	*#attributes*
Movies small		
IMDB4	1.1M	12

Datasets. To evaluate the performance of *BLAST2*, we used two multi-data source scenarios. The datasets employed for the tests have been collected from the Magellan repository [7]; in particular, we consider three collections of heterogeneous records that we called `Movies small`, `Books`, and `Movies big`. `Movies small` is composed of seven datasets that contain records gathered online from `RogerEbert.com`, `Amazon.com`, `RottenTomatoes.com`, and `IMDB.com`, all about movies. `Books` is composed of 10 datasets that contain records gathered online from `BarnesandNoble.com`, `Amazon.com`, `GoodReads.com`, and `Half.com`, all about books. `Movies big` contains the same datasets contained in `Movies small` but extended with another version of IMDB that contains 1.1M of records. Considered singularly, none of these datasets contains duplicates.

The characteristics of the datasets are reported in Table 1. All the considered datasets have different schemas [11]. The ground truth has been generated using the Magellan system [11], and the number of identified duplicates in each scenario is reported in Table 1. The Gold Standard for the attribute clustering has been generated using the MOMIS system as described in Section 4.3 (and reported in Appendix A, Table 2).

4.1 Blast2 Performance

In this experiment, the meta-blocking is applied on block collections generated from the two datasets with different methods:

- *MB NoSchema*: employs the schema-agnostic Token Blocking;
- *BLAST*: employs the Loose-Schema Blocking (i.e., Token Blocking within each attribute partition as described in Section 3.1);
- *BLAST2* employs the LSH in combination with Loose-Schema Blocking;
- *MB SchemaAligned*: employs the Loose-Schema blocking employing the Gold Standard for generating the attribute partitioning. The provided schema perfectly aligns all the attributes (see Section 4.3).

Figure 7 shows the results in terms of recall and precision. All the methods obtain the same recall on `Books` and `Movies small`. On `Movies big` (Figure 7(e)), *BLAST* and *BLAST2* obtain a

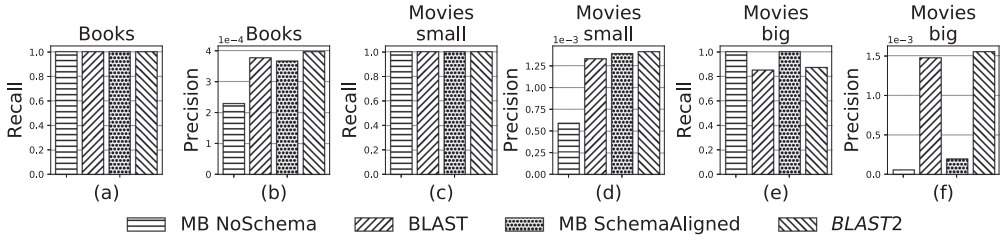

Fig. 7. Precision and recall over all datasets.

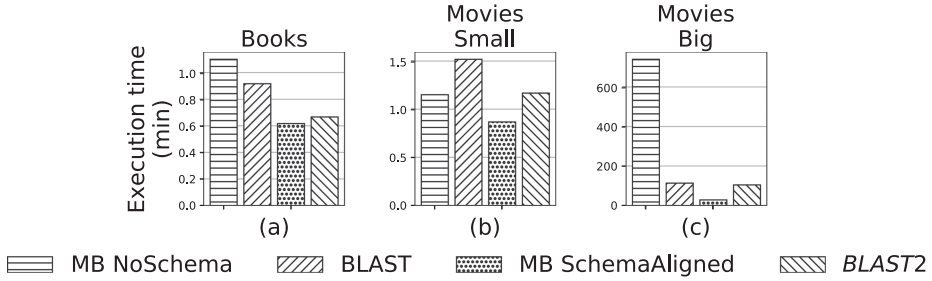

Fig. 8. Execution time of meta-blocking over all datasets.

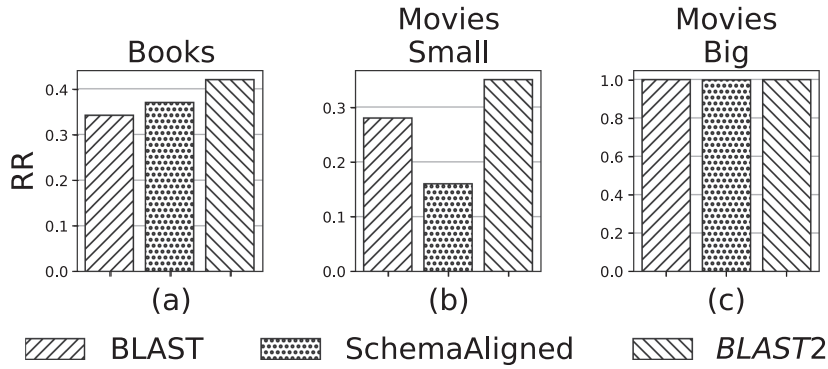

Fig. 9. Reduction Ratio (RR) of meta-blocking over all datasets. The reduction ratio is computed with respect to the number of comparisons generated with Token Blocking.

lower recall than *MB NoSchema* and *MB SchemaAligned*. *BLAST*2 obtains a better precision on all the datasets (Figure 7(b), (d), and (f)), showing the effectiveness of the proposed method.

Figure 8 shows the execution time of all methods on both datasets. `MB SchemaAligned` is always faster than other methods, but we are not considering here the time to generate it, which requires the user in the loop. On all the datasets *BLAST*2 outperforms *Blast*, as it is 27% faster on `Books`, 23% faster on `Movies small`, and 8% faster on `Movies big`.

Figure 9 shows the Reduction Ratio (RR) with respect to the original block collection obtained with Token Blocking. RR expresses the relative decrease in cardinality of the new collection of

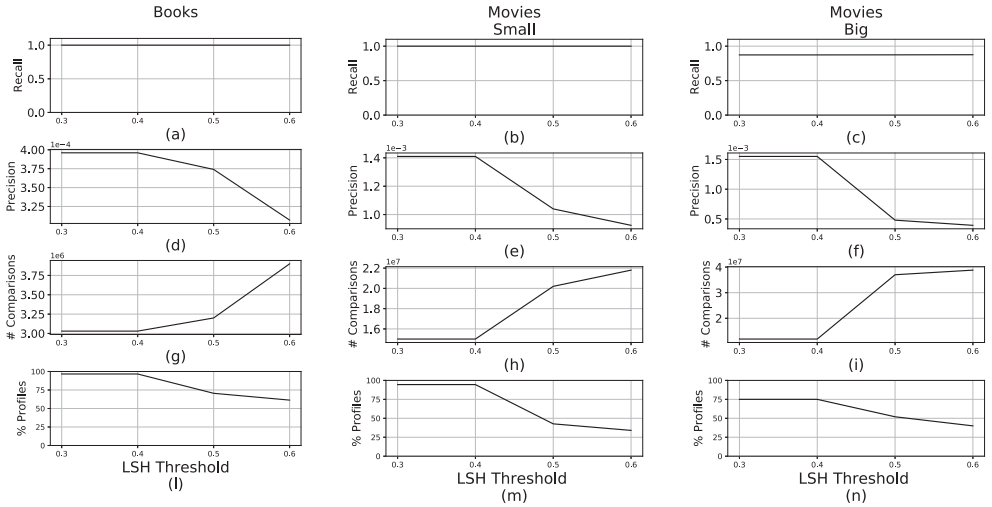

Fig. 10. Precision, recall, number of comparisons, and percentage of profiles used in the schema alignment step, obtained by varying the LSH threshold used in *BLAST2*.

block B' in comparison with the original block collection B and is computed as $RR(B, B') = 1 - ||B'||/||B||$, a higher value means better performance. *BLAST2* obtains a better RR on `Books` and `Movies small` (Figure 9(a) and (b)), gaining a 42% reduction of comparisons on `Books` (Figure 9(a)) and 35% on `Movies small` (Figure 9(b)). Regarding `Movies big`, all the methods obtain almost the same RR, due to the high number of superfluous comparisons that are in the original block collection that are removed by the meta-blocking.

These results can be explained by analyzing the clusters of attributes generated by the different methods. *BLAST2* achieves better results by employing LSH as prefiltering generates better clusters for ER, as it uses only the most similar profiles to extract the attribute partitioning information and removes the noise generated by the less-similar ones. For example, *BLAST* generates the cluster {Rotten1.description, IMDB1.description, Rotten2.summary, IMDB3.movie_name, IMDB2.summary} (cluster 10 in Table 3 in Appendix A for the full list of cluster), in which the movie name is erroneously clustered with the movies summaries. This yields superfluous comparisons, as it compares the titles with the summaries, which are most of the time not related; moreover, the summaries contain many tokens, so generating many superfluous blocks. Differently, *BLAST2* places correctly the title in a cluster with all other titles (Table 3, cluster 3, in Appendix A) and making a cluster with only the movies summaries (Table 3, cluster 11, in Appendix A). Moreover, *BLAST2* produces better clusters than the Gold Standard for ER, because by relying only on the actual values of the attributes it clusters together attributes that are not matched by traditional schema-alignment techniques—or by human experts. For example, in the Gold Standard, the attributes directors and creators are separated into two different clusters (Table 2, clusters 5 and 6, in Appendix A), but they can be correlated as frequently the director is also a creator in the datasets. *BLAST2* generates a single cluster for these two attributes (Table 4, clusters 4, in Appendix A), producing better blocks for ER tasks.

4.2 How Does the LSH Threshold Affect BLAST2 Performance?

In this experiment, *BLAST2* is applied on all the datasets by varying the LSH threshold in the preprocessing phase. Figure 10 shows the results in terms of recall, precision, number of

Mapping Table: MOVIE

MOVIE(globalSource)	a1(a1)	i1(i1)	i2(i2)	i3(i3)	r(r)	rt1(rt1)	rt2(rt2)
ContentRating			ContentRating	ContentRating	pg_rating		ContentRating
Director	director	directors	Director	Director	directors	Director	Director
Duration		duration	Duration	Duration	duration	Duration	Duration
Language						Language	
Rating		movie_rating	Rating	RatingValue	critic_rating	RatingValue	Rating
Release Date				ReleaseDate		Release Date	
Summary			Summary				Summary
Title	title	movie_name	Title	Name	movie_name	Name	Title
actors	star	actors	Cast	Cast	actors	Actors , Cast	Cast
genre		genre	Genre	Genre	genre	Genre	Genre
year	year	year	Year	YearRange	year	Year	Year

Fig. 11. Mapping Table for the MOVIE dataset.

comparisons, and percentage of profiles used for the attribute partitioning task. Keeping too-few profiles (i.e., with an high threshold) causes a reduction of precision (Figure 10(d) and (f)). This happens because the attribute partitioning is done only on the values of the attributes, and reducing the profiles leaves too few values to compare to obtain a good attribute partitioning. The attributes that are not similar to any other attribute (i.e., not clustered) are placed in the *Blob* cluster, causing two effects to be noticed: (i) many superfluous comparisons are generated (Figure 10(g) and (h)), and (ii) the recall does not change (Figure 10(a)–(c)).

4.3 Gold Standard Generation: MOMIS

In this section, we describe how to obtain a Gold Standard for the attribute clustering by using the MOMIS Data Integration System [2]. Given the MOVIE scenario (the BOOK scenario), we perform a data integration process by considering all the datasets (each dataset corresponds to a *local class* in the MOMIS terminology). This process includes a schema alignment step (which will be briefly described below) where semantic similarities among attributes of local classes are automatically detected, similar attributes are grouped together into cluster to form a *global attribute*, and then a *Global/Integrated Class* is obtained. The result is represented by means of a *Mapping Table (MT)*, as shown in Figure 11 for the MOVIE scenario: A column represents a *local class*, i.e., a dataset E_j, and a row represents a global attribute, i.e., a cluster C_i of local attributes; the element $MT[C_i][E_j]$ is the set of attributes of E_j belonging to C_i. As an example, the element `MOVIE[actors][rt1]` = {`Actors`, `Cast`} is the set of attributes of the dataset denoted by `rt1` belonging to the cluster denoted by `actors`.[4]

In the following, we describe the main steps of the MOMIS process for the Mapping Table Generation.

***1. Lexical annotation of local schemata*:** to associate a "meaning" to schema labels (class and attribute names of the local schemata) a semi-automatic annotation with respect to a thesaurus/lexical resource is performed; currently, MOMIS uses both the Wordnet lexical database [13] and domain thesaurus. In other words, to make the meaning of the schema labels explicit, they are mapped in elements of a thesaurus/lexical resource. However, in real-world scenarios, automatic lexical annotation methods have limited performance due to the abundant presence of non-dictionary terms, such as acronyms/abbreviations and compound names. To increase the number of comparable schema labels, we proposed in Reference [23] a schema label normalization method; this methods is implemented in the NORMalizer of Schemata (NORMS) component of MOMIS.

As shown in Figure 12, the first step of the schema label normalization method is a simple *preprocessing step*, where terms are tokenized; for example, the attribute name "pg_rating" is tokenized

[4]Please note that in Figure 11 datasets are denoted by symbols, with the following correspondence: `a1` is `Amazon`.

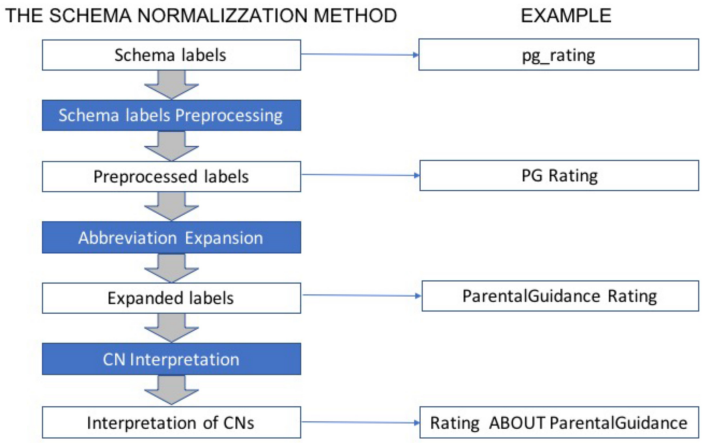

Fig. 12. Overview of the normalization method: for each phase the input and output is shown, with a related example.

in two words, "PG" and "Rating." The second step is the *abbreviation expansion* step, where abbreviations are recognized and then expanded; in our example, the abbreviation "PG" is expanded as "ParentalGuidance."[5] The last, and most important, step is the compound names Interpretation, where the expanded label "ParentalGuidance Rating" is recognized as a compound name, then a semi-automatic method for its interpretation is applied, and a new entry for the label "ParentalGuidance Rating" is inserted in the local Wordnet lexical database. The output of the normalization method is the normalized label "ParentalGuidance Rating" with its interpretation ("Rating ABOUT ParentalGuidance").

***2. Common Thesaurus and Mapping Table Generation*:**

The local schemata are used by MOMIS to construct a Common Thesaurus (CT) consisting of the following semantic relationships among schema labels: synonyms (SYN), holonymy/meronymy (RT), and narrower terms/broader terms/ (NT/BT). Such semantic relationships are mainly lexicon-derived relationships, i.e., relationships derived by the annotation of local schemata with respect to a thesaurus/lexical resource, such as WordNet. For example, a SYN relationship between `cast` and `actor` can be directly derived from Wordnet, thank to the gloss "the actors in a play" of `cast`; moreover, in WordNet, `star` is a direct hyponym of `actor`. However, to find a SYN relationship between `name` and `title` can be derived only after we annotate the term `name` as `title`. This operation is performed only once and then applied to, i.e., the annotation can be transferred to, all attributes `name` of other sources to be integrated. Other semantic relationships are schema-derived relationships, i.e., intra schema relationships derived, for example, from key/foreign keys constraints of a relational source. Moreover, the integration designer can specify domain knowledge by adding specific relationships.

The relationships of the Common Thesaurus are then used to cluster similar *local attributes*, thus building a *global attribute*; for example, in Figure 11, the global attribute `ContentRating` (global attributes are in the first column, with a blue background) is derived from four local attributes. In this way, a global class is generated, with global attributes mapped with the local sources' attributes. For the MOVIE dataset we obtained the Mapping Table shown in Figure 11; the corresponding clusters are shown in Table 2 of Appendix A. Each global attribute corresponds to a cluster (for

[5]In this task the tool uses user-defined or domain-specific Abbreviation Dictionaries, to disambiguate among different interpretations, see, for example, https://www.acronymfinder.com/PG.html.

example, the global attribute `Director` corresponds to cluster 5), and the non-aligned attributes (i.e., global attributes with only a corresponding local attribute) are placed into a BLOB cluster. For a more complete mapping see Table 2.

As a final comment, the construction of the gold standard with MOMIS was effective, as it required the manual annotation of few terms; furthermore, the use of the NORMS component allowed the automatic alignment of various compound names. A similar process was performed to obtain the Gold Standard for the BOOK dataset and for the Movies Big dataset.

5 CONCLUSION AND FUTURE WORK

In this article, we presented *BLAST2*, a novel technique to efficiently extract *loose schema information* (i.e., statistics gathered directly from the data for approximately describing the data sources schemas). We proposed a novel LSH-based preprocessing that improves the *loose schema information* extraction over our previous work. Finally, we experimentally evaluated it on real-world multi-source datasets. The experimental results showed that *BLAST2* outperforms the existing state-of-the-art meta-blocking approaches in terms of quality of the results and execution time.

As a future work, we will explore the proposed method also in the context of the *Schema Matching* problem in a scenario with opaque column names (i.e., when the names of the attributes are not meaningful/not given). The basic idea was proposed in Reference [3], where the authors have formulated an algorithm composed essentially of two steps: In the first step, the duplicates between the datasets (with non-aligned schemes) are identified; in the second step, these duplicates are used to match the schema with opaque column names. This method is limited to the comparison between two schemas, while our attribute clustering approach can be applied with more than two datasets, i.e., applicable in a more general multi-source scenario.

Further, we plan to integrate other sources of "loose" schema information, for instance by extracting approximate functional dependencies among attributes of the datasets [4, 5] to better identify clusters of related attributes.

A APPENDIX

Table 2. Clusters of Attributes Generated with MOMIS

Gold Standard

ID	Attributes
1	IMDB2.duration, RogerErbert.duration, Rotten1.duration, IMDB3.duration, IMDB1.duration, Rotten2.duration
2	Rotten1.year, Rotten2.year, IMDB2.year, IMDB3.year, RogerErbert.year, IMDB1.yearrange, Amazon.year
3	Amazon.star, IMDB2.cast, Rotten1.actors, Rotten2.cast, IMDB3.actors, RogerErbert.actors, Rotten1.cast, IMDB1.cast
4	IMDB1.name, Rotten1.name, RogerErbert.movie_name, Rotten2.title, Amazon.title, IMDB2.title, IMDB3.movie_name
5	RogerErbert.directors, IMDB1.director, Rotten2.director, IMDB2.director, Amazon.director, Rotten1.director, IMDB3.directors
6	IMDB2.creators, Rotten1.creator, IMDB1.creator, Rotten2.creators
7	Rotten2.contentrating, IMDB2.contentrating, RogerErbert.pg_rating, IMDB1.contentrating
8	Rotten1.description, IMDB1.description
9	IMDB1.genre, Rotten2.genre, RogerErbert.genre, IMDB3.genre, Rotten1.genre, IMDB2.genre
10	IMDB1.releasedate, Rotten1.release date
11	Rotten1.ratingvalue, IMDB2.rating, IMDB3.movie_rating, IMDB1.ratingvalue, Rotten2.rating, RogerErbert.critic_rating
12	Rotten2.summary, IMDB2.summary
BLOB	Rotten1.reviewcount, Rotten1.ratingcount, Rotten1.filming locations, Rotten1.language, IMDB1.url, Amazon.cost, Amazon.time, Rotten1.country

Table 3. Clusters of Attributes Generated with BLAST

BLAST

ID	Attributes
0	IMDB2.duration, RogerErbert.duration, Rotten1.duration, IMDB3.duration
1	Rotten1.year, Rotten2.year, IMDB1.releasedate, IMDB2.year, IMDB3.year, RogerErbert.year, IMDB1.yearrange, Amazon.year, Rotten1.release date
2	Amazon.star, IMDB2.cast, Rotten1.actors, Rotten2.cast, IMDB3.actors, RogerErbert.actors
3	IMDB1.name, Rotten1.name, RogerErbert.movie_name, Rotten2.title, Amazon.title, IMDB1.url, IMDB2.title
4	RogerErbert.directors, IMDB1.director, Rotten2.director, IMDB2.director, Amazon.director, Rotten1.director, IMDB3.directors
5	IMDB2.creators, Rotten1.creator, IMDB1.creator, Rotten2.creators
6	Rotten1.cast, IMDB1.cast
7	Rotten1.ratingvalue, IMDB2.rating, IMDB3.movie_rating, IMDB1.ratingvalue, Rotten2.rating, RogerErbert.critic_rating
8	IMDB1.duration, Rotten2.duration, Amazon.cost
9	Amazon.time, Rotten1.reviewcount, Rotten1.ratingcount
10	Rotten1.description, IMDB1.description, Rotten2.summary, IMDB3.movie_name, IMDB2.summary
11	IMDB1.genre, Rotten2.genre
12	Rotten2.contentrating, IMDB2.contentrating, RogerErbert.pg_rating
13	RogerErbert.genre, IMDB3.genre, Rotten1.genre, IMDB2.genre
BLOB	Rotten1.filming locations, Rotten1.country, IMDB1.contentrating, Rotten1.language

Table 4. Clusters of Attributes Generated with *BLAST2*

BLAST2

ID	Attributes
0	Amazon.time, IMDB2.duration, Rotten1.reviewcount, RogerErbert.duration, Rotten1.duration, IMDB3.duration, Rotten1.ratingcount
1	Rotten2.year, IMDB2.year
2	Amazon.star, IMDB2.cast, Rotten1.actors, Rotten2.cast, IMDB3.actors, RogerErbert.actors, Rotten1.cast, IMDB1.cast
3	IMDB1.name, Rotten1.name, Rotten1.filming locations, RogerErbert.movie_name, Rotten2.title, Amazon.title, IMDB1.url, IMDB3.movie_name, IMDB2.title
4	IMDB1.director, IMDB2.creators, RogerErbert.directors, Rotten2.director, IMDB2.director, Rotten2.creators, Amazon.director, Rotten1.director, IMDB3.directors
5	Rotten1.year, IMDB1.releasedate, RogerErbert.year, IMDB3.year, IMDB1.yearrange, Amazon.year, Rotten1.release date
6	Rotten1.ratingvalue, IMDB2.rating, IMDB3.movie_rating, IMDB1.ratingvalue, Rotten2.rating, RogerErbert.critic_rating
7	IMDB1.duration, Rotten2.duration, Amazon.cost
8	IMDB1.genre, Rotten2.genre
9	Rotten2.contentrating, IMDB2.contentrating, RogerErbert.pg_rating
10	Rotten1.creator, IMDB1.creator
11	Rotten1.description, IMDB1.description, Rotten2.summary, IMDB2.summary
12	RogerErbert.genre, IMDB3.genre, Rotten1.genre, IMDB2.genre
BLOB	Rotten1.country, IMDB1.contentrating, Rotten1.language

REFERENCES

[1] Domenico Beneventano, Sonia Bergamaschi, Luca Gagliardelli, and Giovanni Simonini. 2019. Entity resolution and data fusion: An integrated approach. In *Proceedings of the 27th Italian Symposium on Advanced Database Systems, Castiglione della Pescaia (Grosseto'19)*. http://ceur-ws.org/Vol-2400/paper-17.pdf.

[2] Sonia Bergamaschi, Domenico Beneventano, Francesco Guerra, and Mirko Orsini. 2011. Data integration. In *Handbook of Conceptual Modeling: Theory, Practice and Research Challenges*, D. W. Embley and B. Thalheim (Eds.). Springer Verlag.

[3] Alexander Bilke and Felix Naumann. 2005. Schema matching using duplicates. In *Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*. 69–80. DOI: https://doi.org/10.1109/ICDE.2005.126

[4] Loredana Caruccio, Vincenzo Deufemia, and Giuseppe Polese. 2015. Relaxed functional dependencies—A survey of approaches. *IEEE Trans. Knowl. Data Eng.* 28, 1 (2015), 147–165.

[5] Loredana Caruccio, Vincenzo Deufemia, and Giuseppe Polese. 2017. Evolutionary mining of relaxed dependencies from big data collections. In *Proceedings of the 7th International Conference on Web Intelligence, Mining and Semantics*. 1–10.

[6] Peter Christen. 2012. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE Trans. Knowl. Data Eng.* 24, 9 (2012), 1537–1555.

[7] Sanjib Das, AnHai Doan, Paul Suganthan G. C., Chaitanya Gokhale, and Pradap Konda. [n.d.]. The Magellan Data Repository. Retrieved from https://sites.google.com/site/anhaidgroup/useful-stuff/data.

[8] 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, C (2017), 137–157.

[9] Luca Gagliardelli, Giovanni Simonini, Domenico Beneventano, and Sonia Bergamaschi. 2019. SparkER: Scaling entity resolution in Spark. In *Proceedings of the 22nd International Conference on Extending Database Technology (EDBT'19)*.

[10] Simonini Giovanni, Papadakis George, Palpanas Themis, and Bergamaschi Sonia. 2018. Schema-agnostic progressive entity resolution. In *Proceedings of the 21st International Conference on Data Engineering (ICDE'18)*. 53–64.

[11] Pradap Konda, Sanjib Das, Paul Suganthan G. C., AnHai Doan, Adel Ardalan, Jeffrey R. Ballard, Han Li, Fatemah Panahi, Haojun Zhang, Jeff Naughton, et al. 2016. Magellan: Toward building entity matching management systems. *Proc. VLDB Endow.* 9, 12 (2016), 1197–1208.

[12] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. 2014. *Mining of Massive Datasets*. Cambridge University Press.

[13] George A. Miller. 1995. WordNet: A lexical database for English. *Commun. ACM* 38, 11 (1995), 39–41. DOI: https://doi.org/10.1145/219717.219748

[14] George Papadakis, Ekaterini Ioannou, Themis Palpanas, Claudia Niederee, and Wolfgang Nejdl. 2013. A blocking framework for entity resolution in highly heterogeneous information spaces. *IEEE Trans. Knowl. Data Eng.* 25, 12 (2013), 2665–2682.

[15] 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.

[16] George Papadakis, George Papastefanatos, and Georgia Koutrika. 2014. Supervised meta-blocking. *Proc. VLDB* 7, 14 (2014), 1929–1940. DOI: https://doi.org/10.14778/2733085.2733098

[17] George Papadakis, George Papastefanatos, Themis Palpanas, and Manolis Koubarakis. 2016. Scaling entity resolution to large, heterogeneous data with enhanced meta-blocking. In *Proceedings of the 22nd International Conference on Extending Database Technology (EDBT'16)*. 221–232.

[18] George Papadakis, Leonidas Tsekouras, Emmanouil Thanos, George Giannakopoulos, Themis Palpanas, and Manolis Koubarakis. 2020. Domain-and structure-agnostic end-to-end entity resolution with JedAI. *ACM SIGMOD Rec.* 48, 4 (2020), 30–36.

[19] Thilina Ranbaduge, Dinusha Vatsalan, and Peter Christen. 2018. A scalable and efficient subgroup blocking scheme for multidatabase record linkage. In *Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 15–27.

[20] Pavel Shvaiko and Jérôme Euzenat. 2013. Ontology matching: State of the art and future challenges. *IEEE Trans. Knowl. Data Eng.* 25, 1 (2013), 158–176.

[21] Giovanni Simonini, Sonia Bergamaschi, and H. V. Jagadish. 2016. BLAST: A loosely schema-aware meta-blocking approach for entity resolution. *VLDB Endow.* 9, 12 (2016), 1173–1184.

[22] Giovanni Simonini, Luca Gagliardelli, Sonia Bergamaschi, and H. V. Jagadish. 2019. Scaling entity resolution: A loosely schema-aware approach. *Inf. Syst.* 83, 7 (2019), 145–165.

[23] Serena Sorrentino, Sonia Bergamaschi, Domenico Beneventano, and Laura Po. 2010. Automatic normalization and annotation for discovering semantic mappings. In *Search Computing—Trends and Developments*, Lecture Notes in Computer Science, Vol. 6585. Springer, 85–100.

Received October 2019; revised March 2020; accepted April 2020