Short paper

Boosting Blocking Performance in Entity Resolution Pipelines: Comparison Cleaning using Bloom Filters

Leonardo Gazzarri
University of Stuttgart
Stuttgart, Germany
leonardo.gazzarri@ipvs.uni-stuttgart.de

Melanie Herschel
University of Stuttgart
Stuttgart, Germany
melanie.herschel@ipvs.uni-stuttgart.de

ABSTRACT

Entity Resolution (ER) allows to identify different virtual representations of entities that refer to the same real world entity. When applied to highly heterogeneous data, ER relies on schema-agnostic blocking techniques to improve efficiency while yielding good effectiveness. A drawback of schema-agnostic blocking is the potentially high number of redundant pairwise comparisons. This has led to the introduction of additional efficiency layers beyond blocking in the overall ER pipeline, which all aim at pruning comparisons to reduce the unnecessary time overhead.

This paper proposes a novel technique based on Bloom filters that integrates in such an efficiency layer. In addition to avoiding redundant comparisons, it further prunes superfluous comparisons that are unlikely to result in matches when actually compared. Experiments on benchmark datasets show that our approach improves existing approaches in space and time efficiency, with insignificant changes in effectiveness.

1 INTRODUCTION

Entity resolution (ER) is the problem of identifying or *matching* different digital representations of the same real-world entity (e.g., the same person, manufactured part). It represents a fundamental task in data integration and data cleaning. While ER has mostly been studied for homogeneously structured data (to which we refer to as *structured ER*) [6], recent work has been extended to *unstructured ER*, i.e., ER when it is not possible or useful to transform heterogeneous entities to match a common schema [5, 10]. This for instance applies when considering ER for the Web of Data or for data stored in data lakes.

For large datasets, handling the inherently quadratic complexity of ER generally becomes computationally prohibitive. Therefore, ER solutions commonly adopt blocking techniques [6, 12] that reduce the total number of pairwise comparisons by performing comparisons only between entity representations placed in the same block according to some criteria. This typically prunes a significant number of comparisons between entity descriptions that do not match anyway, to which we refer to as *superfluous* comparisons. In structured ER, blocking techniques (and ER in general) heavily rely on a fixed schema among all entity representations. As this assumption does not hold in unstructured ER, schema-agnostic blocking techniques have recently been proposed for unstructured ER [5, 8, 10].

Figure 1 depicts a general pipeline for unstructured ER [12]. It comprises three layers enabling efficient and effective ER. *Block building* places entity representations, each denoted as e_i, into blocks. One problem of schema-agnostic blocking is that the

© 2020 Copyright held by the owner/author(s). Published in Proceedings of the 23rd International Conference on Extending Database Technology (EDBT), March 30-April 2, 2020, ISBN 978-3-89318-083-7 on OpenProceedings.org.
Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

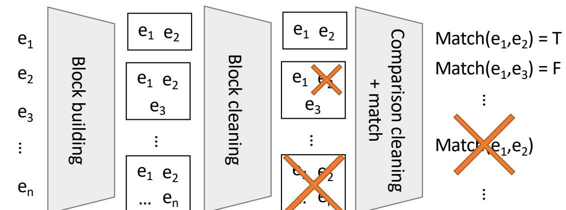


Figure 1: Unstructured ER pipeline.

resulting blocks may significantly overlap and thus yield *redundant* comparisons and the distribution of entity descriptions over blocks may still yield too many pairwise comparisons. These two problems have resulted in two further techniques for unstructured ER. *Block cleaning* acts at a block level and either prunes entire blocks (e.g., too large and thus most likely resulting in mostly superfluous comparisons) or entity descriptions within blocks (e.g., descriptions appearing in too many blocks are removed from the least important blocks). Finally, *comparison cleaning* considers pairs of entity descriptions resulting from the cleaned block collection. It prunes pairs if they are identified as either redundant or superfluous. Otherwise, pairs of entity descriptions are compared using a *match* function to determine whether they represent the same entity or not. Examples of comparison cleaning techniques are *comparison propagation* [9] and *meta-blocking* approaches [4, 11, 14].

Contribution. This paper presents a novel comparison cleaning approach that prunes both redundant and superfluous comparisons. It relies on (i) a favorable order of blocks being processed, for which we validate a heuristic that works in practice, and (ii) false positives that are, in our context, a useful feature of *Bloom filters (BFs).* Indeed, while false positives are typically undesired, we shall see that we can turn them to our advantage when coupled with a favorable order. As Bloom filters are static data structures that need to be correctly set up upfront, we further extend our method to use *scalable Bloom filters (SBFs).* Our experimental validation on several real-world benchmark datasets shows that comparison cleaning using SBFs is robust to different block collection characteristics and provides a good trade off between efficiency and effectiveness of comparison cleaning that improves on baseline and state-of-the-art solutions.

Structure. Section 2 introduces our novel approach based on Bloom filters and its extensions. Section 3 presents our experimental evaluation. We conclude in Section 4.

2 BLOOM FILTER ENHANCED BLOCKING

BFs have been previously used in ER, for example, to obscure sensitive data [15] or to summarize the blocking structure of a dataset [7]. In our work, we employ BFs for comparison cleaning. In this section, we describe our algorithm leveraging BFs (Section 2.1), the block ordering heuristic allowing to turn false positives inherent to BFs to our advantage (Section 2.2), and details on extending our algorithm with SBFs (Section 2.3).

Algorithm 1: Comparison cleaning using a Bloom filter

```
Input: p_max, s, B
1 initialize(BF, p_max, s);
2 while B has further pairs to process do
3     p_{i,j} ← next pair according to order defined by P;
4     if !lookup(BF, key(i, j)) then
5         insert(BF, key(i, j));
6         submit(match(p_{i,j}));
```

2.1 General BF based comparison cleaning

To discuss our algorithm that uses BFs, we first provide relevant background on BFs. We refer readers to [2, 3] for further details.

A BF is a space-efficient probabilistic data structure that offers two operations: *insert(k)* inserts the key k in the filter and *lookup(k)* answers if the key k has been inserted with some probability, or definitely if it has not been inserted. False positive probability (the probability of a lookup returning true even though k has not been inserted) can be computed and it increases with insertions of unique keys. A Bloom filter BF is initialized by providing a maximum acceptable false positive probability p_{max} and the number of expected keys to be inserted, denoted s.

In our context, a key $k_{i,j}$ uniquely identifies a pair $p_{i,j} = (e_i, e_j)$. We assume that e_i and e_j are unique identifiers of entity descriptions. We generate $k_{i,j}$ using a function $key : Integer \times Integer \rightarrow Integer$. The pairs to be sequentially inserted into the BF (by inserting their integer key) are pairs produced from a block collection $B = [b_1, \ldots, b_n]$ that comprises n blocks of various sizes. The maximum number of keys possibly being inserted is bounded by $O(|b_1|^2 + |b_2|^2 + \ldots + |b_n|^2)$. Given that the false positive probability increases with the number of keys that have been inserted into a BF, we assume that we can iterate over pairs resulting from B in a specific order. That is, our algorithm uses an iterator to retrieve pairs in the order given by

$$P = \left[(e_i^k, e_j^k) \middle| e_i^k e_j^k \in b_k, 1 \leq k \leq n, 1 \leq i < |b_k|, i < j \leq |b_k| \right]$$

where the three ranges specified for k, i, and j are to be interpreted as three nested loops with k being the outer and j the most inner loop. How to sort B in a good order for efficient and effective comparison cleaning is further discussed in Section 2.2.

Using the above assumptions and notation, Algorithm 1 summarizes how we leverage BFs to perform comparison cleaning. It first initializes a Bloom filter BF, given p_{max} and s. While p_{max} is generally user defined (we will see how to best set it in practice in Section 3), s can also be computed as the upper bound of comparisons based on B (which we do in all experiments). The algorithm then iterates over pairs retrieved from B in the order given by P, retrieving each pair $p_{i,j}$ one at a time. In line 4, we check if $p_{i,j}$ has already been inserted in BF. To this end, a unique key $k_{i,j}$ is generated for pair $p_{i,j}$. If looking up $k_{i,j}$ in BF returns false, we know that the pair has not been compared before. So we insert the key into BF, perform the pairwise comparison of the actual pair of entity descriptions using a *match* function, and propagate the result (could be e.g., a boolean, a similarity score, or a match probability) for further processing via a submit method.

Time analysis. Time to insert or search a key in a BF is $O(|H|)$, with $|H|$ the number of hash functions used to fingerprint the key in the BF [3]. Defining $\bar{c}$ the average cost for comparing two entities, the desiderata is that $O(|H|) \ll \bar{c}$. This typically holds in practice because hashing a pair of integers is less expensive than fetching and matching two entity representations.

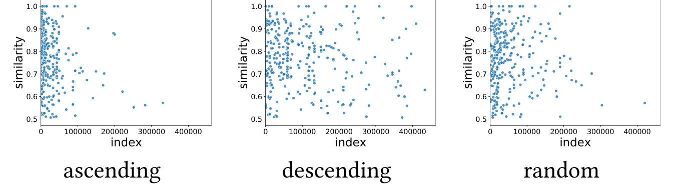


Figure 2: Distribution of pairwise similarities per iteration for different order heuristics

Space analysis. The minimum bits size m of the filter is determined by $m \geq 1.44s \cdot log_2(1/p_{max})$ [3]. For instance, assuming $p_{max} = 0.1$, we have $m \approx 4.78s$ bits.

In our algorithm, we can easily replace the BF by any other dictionary data structure implementing an $insert(\cdot)$ and a $lookup(\cdot)$ method. We opt for BFs for two reasons. First, a BF is a well known space efficient data structure and we validate experimentally that in comparison to other dictionary data structures, it exhibits better performance for comparison cleaning (see Section 3). Second, a BF allows us to additionally prune superfluous comparisons. This slightly more hidden benefit is rooted in both p_{max} inherent to Bloom filters and the order of entity pairs determined by P. This interplay is further discussed next.

2.2 Block ordering heuristic

The false positive probability inherent to a BF results in three possible cases when a pair $p_{i,j}$ is processed by Algorithm 1:

- **True negative (*tn*)** $lookup(BF, p_{i,j})$ returns false, so $p_{i,j}$ is not redundant
- **True positive (*tp*)** $lookup(BF, p_{i,j})$ returns true and the pair is indeed redundant
- **False positive (*fp*)** $lookup(BF, p_{i,j})$ returns true even though $p_{i,j}$ has not yet been inserted

While *tn* and *tp* are desired cases, *fp* may be problematic for the effectiveness of ER as it may reduce the recall of ER if a wrongly pruned $p_{i,j}$ had, if compared, resulted in a match. We refer to this case as *miss-match*. On the other hand, if $p_{i,j}$ would have resulted in a non-match upon comparison, the comparison was a superfluous one, i.e., a comparison we actually want to prune.

When the number of keys inserted in the BF increases, the false positive rate of the BF increases up to p_{max}. The basic idea of our Bloom filter based comparison cleaning is to put this "effect" of BFs to good use by determining an order of pairs for P that has a high probability of processing true matches early (to minimize the negative effect of miss-match cases) and using the increasing false positive probability coming with later processing to prune superfluous comparisons. Such behavior is intuitively obtained when finding an order of pairs where the match probability decreases with increasing number of iterations. Clearly, the performance of our approach relies on both identifying a suited order that mimics the behavior described above, and on the parameter p_{max}. Given a block collection, [12] postulated that similar entity descriptions are more likely to be found in small blocks rather than in big blocks. We experimentally validate this heuristic on several real-world datasets (see Table 1). Figure 2 shows representative results for three order heuristics resulting from sorting blocks in ascending, descending, and random order of the number of entities they contain on the CDDB dataset. The plots show the first occurrence of a pair $p_{i,j}$ at position (x, sim) when it is the x-th pair to be processed and the similarity computation yields a similarity sim. We cut off similarities below 0.5 for better readability. As is common in ER, we assume that the match

Dataset	Size(D1)	Size(D2)	Duplicates	Brute-force
AG Products	1354	3039	1104	4.11e06
CdDb	9763	//	299	4.77e07
Movies	27615	23182	22813	6.40e08

Table 1: Dataset characteristics (same as in [1]).

	Dataset	Redundancy	Comparisons	Recall
E1	AG Products	79%	1.9e7	1.0
E2	Movies	15%	6.5e7	0.98
E3	Movies	7%	9.7e6	0.96
E4	CdDb	36%	2.2e7	0.99
E5	CdDb	7%	4.6e5	0.99

Table 2: Experiment settings.

function determines $p_{i,j}$ to match if its similarity is above a similarity threshold. So the higher the similarity, the more likely it is we determine a match. Clearly, the ascending order of block sizes best mimics the desired behavior for our order heuristic. This experimentally validates the claim that in practice, sorting the block collection B in ascending order of its block sizes is suited to approximate the desired behavior on match probability.

2.3 Extension using scalable Bloom filters

We have seen in Section 2.1 that initializing BF requires setting both p_{max} and s, which are key in optimally setting the number of bits allocated to the Bloom filter. While s is in the order of sum of squares of individual block sizes, this is in practice a very loose upper bound for the expected number of *unique* keys to be inserted, especially when a high degree of redundancy can be expected. To avoid allocating unnecessary space to a BF and be less sensitive to variations of both the redundancy and match distribution in P, we explore how to extend our comparison cleaning algorithm with scalable Bloom filters. We provide relevant background on SBFs and refer readers to [2] for details.

A SBF is a list of Bloom filters $BF_0, \ldots, BF_n$. Initially, the list includes a single BF, denoted BF_0, with an associated initial capacity s_0 and maximum false positive probability p_0. More generally, each BF_i has an associated capacity s_i and a false positive probability p_i. Given BF_i the last BF in the list, a new key is inserted into BF_i. When BF_i becomes full (i.e., new keys cannot be inserted without exceeding p_i), a new BF_{i+1} is inserted in the list. Given a tightening ratio θ with $0 < \theta < 1$ and a growth ratio $\sigma \geq 1$, BF_{i+1} is set with $p_{i+1} = p_0\theta^i$ and $s_{i+1} = s_i\sigma$. Overall, the capacity of the SBF gradually increases as needed while the compound false positive probability p_{fp} is bounded by $p_{fp} \leq p_0 \frac{1}{1-\theta}$. This means that we do no longer have to set s upfront, assuming the worst case in Algorithm 1. We can choose a more conservative initial capacity $s_0 << s$ to improve space efficiency and extend it if needed. The additional parameters of SBF can be fixed to $\theta = 0.9$ and $\sigma = 2$, following the recommendation of [2].

3 EVALUATION

We experimentally validate the comparison cleaning approaches presented in this paper. We both perform a parameter sensitivity study and a comparative evaluation to baseline and state-of-the-art methods. Performance metrics are runtime (time), memory footprint (space), and quality (recall over the set of executable comparisons considering a ground truth file).

All algorithms were implemented in Java 1.8 as extensions of the JedAI library [13] that supports numerous state-of-the-art unstructured ER solutions. We ran experiments on an OpenStack virtualized server (16 processors at 2.30GHz, 50GB RAM).

Our experiments use data from three benchmark datasets commonly used to evaluate unstructured ER (e.g., in [8, 11]). Their characteristics are summarized in Table 1. They are publicly available at the JedAI webpage together with ground truth files.

Our evaluation relies on different experiment settings. Each setting varies in the block collection B input to the evaluated comparison cleaning techniques, obtained by varying datasets and steps preceding comparison cleaning. Table 2 summarizes the characteristics of five settings E_1 through E_5. Here, "Redundancy" reports the fraction of redundant pairs produced by B. "Comparisons" is the total number of pairs resulting from B, i.e., $|P|$. "Recall" reports the maximum possible recall that can be obtained based on B.

3.1 Parameter sensitivity

We study the effect of parameter variations on performance by varying the parameters for both BF and SBF as follows:

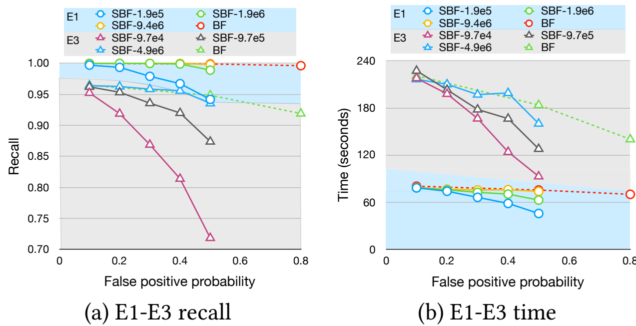
BF Capacity $s = |P|$ (see Comparison column in Table 2), $p_{max} \in \{0.1, 0.5, 0.8\}$
SBF Initial capacity $s_0 \in \{0.01|P|, 0.1|P|, 0.5|P|\}$ and $p_0 \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$

For all considered parameter settings, we studied their effect on time, space, and recall. Due to space constraints, we only report representative results on settings E_1 and E_3, two extreme cases with high and low redundancy. The remaining results are analogous and in line with our discussion.

Figure 3(a) studies recall when varying the false positive probability for different configurations of the capacity in both E_1 and E_3. For setting E_1, we observe that recall is stable across all tested configurations. The reason is that E_1 has high redundancy, so even the most aggressive configuration setting for a SBF (low initial capacity, high initial false positive probability) has an acceptable low loss in recall because typically, it only prunes comparisons that are indeed unnecessary. In E_3, where we have low redundancy, an aggressive solution based on SBFs with a very small capacity and high false positive probability loses much more in effectiveness (25% total loss in recall) compared to a solution using an higher capacity and same false positive probability or a solution that uses a smaller false positive probability. This is due to the fact that a SBF with such aggressive configuration converges faster to the maximum compound false probability than the other solutions, resulting in more miss-matches.

Figure 3(b) shows runtime for the same configurations as the previous experiment. In E_1, the aggressive configuration mentioned before using SBF with very low initial capacity and high false positive probability is particularly efficient because it starts to remove non-redundant pairs earlier than the other solutions but in a point where the match probability is already very low. This positive effect on runtime decreases as the initial capacity of a SBF increases, until it eventually converges to the behavior of a BF. In E_3 and similarly in other settings with low to moderate redundancy, the behavior is similar, yet the slopes are more evident. This is due to the fact that in such settings, higher false positive probabilities prune more pairs that are not redundant (and thus pruned by any configuration).

Considering space (no graphs shown for conciseness), all configurations require less than 25MB of memory, making both SBFs and BFs space efficient data structures. Considering BFs, as expected by the formula in Section 2, the space decreases with



(a) E1-E3 recall

(b) E1-E3 time

Figure 3: Recall and runtime for parameter configurations

increasing p_{max}. Considering SBFs, we observe that required space is lower than for BFs only when the degree of redundancy is sufficiently high.

From our parameter sensitivity analysis, we observe that good performance across all settings is obtained when SBF capacity is set between 0.1 and 0.5 times $|P|$ (the size of a BF) coupled with a high/moderate false positive probability (0.3 to 0.5). As BFs do not suffer from efficiency or space degradation caused by SBF extensions, their false positive probability can be set to even higher values (as these high values kick in late in the process where typically, most matches have already been processed), for which we recommend the range between 0.5 and 0.8.

3.2 Comparative evaluation

Given our above conclusion, we now select two good configurations: BF* with $s = |P|$, $p_{max} = 0.5$ and SBF* with $s_0 = |P| * 0.5$, $p_0 = 0.5$). We compare them to two other comparison cleaning approaches. The first baseline uses a hash set (HS) instead of a BF. The second approach applies comparison propagation (CP) [9]. CP uses inverted indexes to avoid redundant comparisons. The inverted index is basically a hash-map where the keys are entity identifiers and the list of values associated to the key identifies the block indexes where the entity appears. Given a processing order of the blocks $b_1, b_2, ..., b_n$, two entities are compared in a block b_i only if the lowest common index of their associated block indexes is i. We again evaluate time, space, and recall. We do not consider meta-blocking [11] as a competitor because it is not integrable in Algorithm 1, which interleaves pairwise similarity computation with comparison cleaning.

Quality comparison. Both HS and CP yield the maximum possible recall (see Table 2) as they exclusively prune redundant comparisons. The possibly lower recall of our Bloom filter based solutions is attributed to miss-matches caused by false positives (see Section 2). Throughout all settings E_1 through E_5, the recall of BF* (SBF*) does not reduce by more than 4%.

Time comparison. Table 3a shows the runtime for the compared approaches in all setings. We further report on the runtime of ER without any comparison cleaning in the "Nothing" column. When high redundancy occurs like in E1, all the comparison cleaning solutions outperform the approach that compares all redundant pairs (Nothing). Also, we observe that HS and CS consistently have comparable runtimes. In scenarios with low redundancy, HS and CS are worse than (or comparable to) Nothing, as their overhead time to find redundant pairs is higher than the time avoided by not comparing them (see E_3 and E_4). Both BF* and SBF* outperform the baseline approaches thoughout all settings, improving runtime between 7% and 31% over the best competitor.

	BF*	SBF*	CP	HS	Nothing
E1	75	74	95	81	540
E2	1235	1161	1557	1566	2038
E3	183	160	255	218	221
E4	77	71	103	112	120
E5	4	4	5	5	4

(a) Time (in seconds)

	BF*	SBF*	CP	HS	Dataset
E1	6	3	4	223	3
E2	23	38	55	5063	40
E3	3	5	39	785	40
E4	7	13	12	1281	6
E5	<1	<1	5	39	6

(b) Space (in MB)

Table 3: Results for time and space

Space comparison. Table 3b shows space required by the compared comparison cleaning approaches. Unsurprisingly, HS performs worst. CP performs better because the size of its inverted index depends on the number of entity descriptions of the datasets, the number of blocks in B, and the level of redundancy of entities in multiple blocks. We observe that the space used by BF* and SBF* further usually improves on CP (up to 90%) while, at the same time, improving efficiency and maintaining high quality.

4 CONCLUSIONS

We proposed a novel approach comparison cleaning approach in entity resolution based on Bloom filters that removes both redundant and superfluous comparisons to improve efficiency. The technique relies on a validated heuristic that pairs a decreasing match probability with an increasing false positive probability. We further present an extension to our approach, using scalable bloom filters. Our experimental validation demonstrates that our approach outperforms state-of-the art algorithms in space and time, while maintaining high effectiveness.

ACKNOWLEDGMENTS

This research was partially funded by the Ministry of Science of Baden-Württemberg, Germany, for the Doctoral Program 'Services Computing'.

REFERENCES

[1] [n.d.]. JedAI. https://github.com/scify/JedAIToolkit/, accessed 29.11.2019.
[2] P. Almeida, C. Baquero, N. Preguiça, and D. Hutchison. 2007. Scalable bloom filters. *Inform. Process. Lett.* 101 (2007), 255–261.
[3] A. Broder and M. Mitzenmacher. 2004. Network applications of bloom filters: A survey. *Internet mathematics* 1 (2004), 485–509.
[4] V. Efthymiou, G. Papadakis, G. Papastefanatos, K. Stefanidis, and T. Palpanas. 2017. Parallel meta-blocking for scaling entity resolution over big heterogeneous data. *Information Systems* 65 (2017), 137–157.
[5] V. Efthymiou, K. Stefanidis, and V. Christophides. 2016. Benchmarking blocking algorithms for web entities. *IEEE Transactions on Big Data* (2016).
[6] A. Elmagarmid, P. Ipeirotis, and V. Verykios. 2007. Duplicate record detection: A survey. *TKDE* 19 (2007), 1–16.
[7] D. Karapiperis, A. Gkoulalas-Divanis, and V.S. Verykios. 2018. Summarization Algorithms for Record Linkage.. In *EDBT*.
[8] G. Papadakis, E. Ioannou, C. Niederée, and P. Fankhauser. 2011. Efficient entity resolution for large heterogeneous information spaces. In *WSDM*.
[9] G. Papadakis, E. Ioannou, C. Niederée, T. Palpanas, and W. Nejdl. 2011. Eliminating the redundancy in blocking-based entity resolution methods. In *JCDL*.
[10] G. Papadakis, E. Ioannou, T. Palpanas, C. Niederee, and W. Nejdl. 2013. A blocking framework for entity resolution in highly heterogeneous information spaces. *TKDE* 25 (2013), 2665–2682.
[11] G. Papadakis, G. Koutrika, T. Palpanas, and W. Nejdl. 2014. Meta-blocking: Taking entity resolutionto the next level. *TKDE* 26 (2014), 1946–1960.
[12] G. Papadakis, J. Svirsky, A. Gal, and T. Palpanas. 2016. Comparative analysis of approximate blocking techniques for entity resolution. *PVLDB* 9 (2016), 684–695.
[13] G. Papadakis, L. Tsekouras, E. Thanos, G. Giannakopoulos, T. Palpanas, and M. Koubarakis. 2018. The return of jedAI: end-to-end entity resolution for structured and semi-structured data. *PVLDB* 11 (2018), 1950–1953.
[14] G. Simonini, S. Bergamaschi, and H.V. Jagadish. 2016. BLAST: a loosely schema-aware meta-blocking approach for entity resolution. *PVLDB* 9 (2016), 1173–1184.
[15] D. Vatsalan, P. Christen, and E. Rahm. 2016. Scalable privacy-preserving linking of multiple databases using counting bloom filters. In *ICDMW*.