Blocking music metadata from heterogenous data sources

Oliver Pabst
FG Datenbanken und Informationssysteme
Institut für Praktische Informatik
Leibniz Universität Hannover
pabst@dbs.uni-hannover.de

Udo W. Lipeck
FG Datenbanken und Informationssysteme
Institut für Praktische Informatik
Leibniz Universität Hannover
ul@dbs.uni-hannover.de

ABSTRACT

Entity resolution or object matching describes the assignment of different objects to each other that describe the same object of the real world. It is used in a variety of technical systems, e.g. systems that fuse different data sources. Blocking is used in this context as an approach to reduce the total amount of comparisons by grouping similar objects in the same cluster and dissimilar objects in different clusters. As a result only the objects of the same clusters have to be compared to each other. To deal with noise, for instance spelling errors, that can result from different heterogeneous data sources, various blocking approaches exist that may add or remove redundancy to the data.

In this paper we propose a system that utilizes a derivative of the standard blocking technique to compute correspondences between objects as starting points for a graph matching process. The blocking technique, which usually relies on identity of blocking keys derived from attributes, is modified to cope with heterogenous source data with few attributes suitable for matching. A common criticism of standard blocking is low efficiency, since the block sizes are unbalanced with regard to the number of contained entities. We take precautions to keep the efficiency high by reducing the size and amount of large partitions.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Blocking, matching, entity resolution

Keywords

Blocking, matching, entity resolution

1. INTRODUCTION

Merging two different relational databases is a difficult task. There are well known research approaches in the area of information integration where mappings between relational schemas are the groundwork before relational schemas can be matched and later fused. Whereas relational databases have to compute expensive joins over relations to gather data that are related, the graph data model is predestined to directly store relational data and process them object-wise. While we have to perform complex operations in relational systems to get the join partners of a specific tuple, in the graph data model we just have to query the adjacent edges and nodes of a specific node. Now if we want to use graphs to match entities, we do need some starting points, since testing all possibilities is infeasible.

As an origin we have two relational databases, Discogs[1] and Musicbrainz[2], which collect and maintain music metadata from web communities, that we want to fuse into one database. Both databases are both structurally and syntactically heterogenous. To overcome this heterogeneity, we have defined an integrated schema and have transformed both databases into this schema. There are, however, actually very few attributes in the data that are suitable for an attribute-based matching process. For example, the position of a track on a release or the genre of a release are by no means suitable: Genres occur frequently and possess only a small number of distinct values, i.e., a low entropy. And track positions are not comparable; two tracks are not more similar if the difference of the positions is low. Additionally these data are afflicted by noise in the form of spelling mistakes and modifications by internal processes. No matter how a later object matching is implemented, as the focus of the later object matching could be shifted from attribute-based to relational similarities[5], the blocking before matching has to stay reliable even for few and 'noisy' attributes. Thus we need correspondences between blocks based on similar instead of identical blocking keys. These correspondences form partitions of matching candidates that can be processed in parallel.

During initial tests we have looked at different solutions to perform the blocking process directly on the database system that holds our data. [2] provides a list of possible approaches like q-gram-indexing which first seemed promising, as PostgreSQL already has indexing structures that support q-grams and matching operators in place. Unfortunately the approach did not perform well at all. If it is used to perform string matching between two different tables, a nested loop join is required to compute the string similarities. This leads to very high runtimes (days) for the calculations, despite the

30th GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 22.05.2018 - 25.05.2018, Wuppertal, Germany.

[1]https://www.discogs.com
[2]https://musicbrainz.org

indexes. Additionally the database approach does not scale well at all, since operations are performed sequentially, not in parallel. Other indexing techniques like phonetic codes were considered, but also discarded, because they delivered no suitable results.

To deal with this problem, we have decided to modify the standard blocking technique, first presented by Fellegi and Sunter [3] and later picked up by [9] and [8] to cope with very few available attributes. With regards to performance we want to remain as efficient as standard blocking while decreasing the redundancy and increasing the quality.

This paper is structured as follows. In the next subsection we provide a brief overview of the various classes of blocking techniques. In section 2 we outline the steps of our envisioned process that takes two input data sources and computes starting points for a graph matching process. We explain our new blocking technique and outline the differences to the standard blocking and explain our modifications. Furthermore, we describe the block matching which calculates the correspondences. We show which steps we take to handle noise in the data sources and how we compute block pairs that might be similar and have to be checked in a subsequent graph-matching process; that later process is not part of this paper. In section 3 we describe our experiences with the blocking process. Finally in section 4 we present the conclusions and future work.

Related work

Blocking has been introduced as an important preparation of matching by, e.g., Christen [1]. Over the time a variety of blocking techniques have evolved. Papadakis [7] categorises them into two classes: redundancy-free methods and methods manipulating redundancy. Redundancy in this case means, that the same entity may occur in more than one block. The standard blocking proposed by Fellegi at al. [3] and later modified by Papadakis [8] assigns a blocking key to every entity and groups two entities in the same block if their blocking keys match exactly. This technique is efficient but can lead to insufficient results as it does not handle noise and missing values well, but is free of redundancy.

The other class, methods manipulating redundancy, i.e. methods that may decrease or increase the redundancy, can roughly be separated into three categories of *negative redundancy*, *neutral redundancy* and *positive redundancy*. An example for negative redundancy is canopy clustering [6]. In every iteration an object is picked as a cluster center, points within a threshold distance are assigned to the cluster, and objects inside a smaller threshold distance are completely removed from the dataset. This is justified by the assumption that highly similar objects are most likely in the same block and will not match with other objects. By removing them from the dataset these data cannot be assigned to a new cluster, thus the redundancy is decreased. Redundancy neutral techniques comprise the same number of common blocks across all entity pairs. An example for this category is sorted neighbourhood blocking [4] as the sliding window is shared among all entity pairs of the dataset. Redundancy positive techniques, however, share the concept that the similarity of two entities correlates with the number of blocks that contain both entities. An example is q-gram blocking, where two entities are most similar to each other if they share all q-grams while they are least similar if they have no common q-gram.

Our proposed method at first behaves like a redundancy-positive method since similar (i.e. candidate) entity pairs may appear in multiple block correspondences, but finally it delivers a redundancy-free blocking since duplicate entity pairs are filtered after the block building process.

2. PROCESS OVERVIEW

As described earlier, we want to compute possible correspondences between nodes of two different graphs to initialise a graph-matching process. To achieve this goal we have developed a process that takes data from two sources in the form *s1: (id, string), s2: (id, string)* that is sufficient to describe an entity. In our prototype, that currently works with music metadata, we use the name of artists (together with their local identifier) to approximately identify real world objects in our data sources. Other attributes of artists are not given, but rich contexts of related tracks, labels, releases, etc.

These input data need to be preprocessed since they are afflicted by noise, be it human mistakes like spelling errors which result in transposed characters, omitted strings or modifications by internal processes of the systems [3] that handle the metadata. The string content from an incoming record is split apart at blanks into tokens and the tokens are then processed by applying a stop word list and regular expressions.

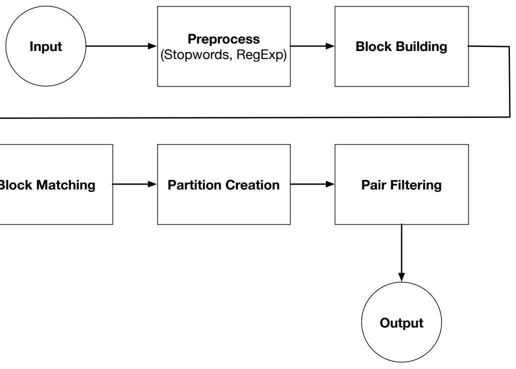

Figure 1: process overview

To reduce the number of comparisons needed to compute possible correspondences between both datasets, we use the standard blocking technique. We work with either of two different strategies to create blocks from the preprocessed input data. The *split-at-blank*-approach leaves the tokens apart, which potentially leads to more than one block per entity. Alternatively blanks are eliminated by the *merge-at-blank*-approach, so that the incoming tokens are merged; with this strategy only one block per entity will be created. Of course, created blocks will collect further entities with the same split/merged tokens.

Then we have to match blocks of entities from our two sources. To avoid missing matches due to noise we do not use an exact matching to compute correspondences between blocks. Instead blocks with same or similar labels shall be matched by applying a string similarity measure on the labels of all block pairs. Thus we can additionally apply a

[3]By system we mean internal rules inside a processing system that affect the data as they are fed into the information system, processed and stored.

variable threshold that can be set appropriately to fit the noise of the data sources.

The resulting matches from the block matching are then used to create partitions. Each match is used to build a partition, which clusters the identifiers of similar block labels together according to the computed block matches. Each match is used to build such a partition by carrying two lists that contain the identifiers participating in the block matching.

To reduce redundancy and thus increase efficiency, we apply a pair filtering after the block matching process. If we consider the strategy that splits the entity string at blanks into multiple tokens, it is obvious that an entity can be contained in multiple blocks. Since this applies to both data sources, it is clear that the same entity can participate in more than one candidate pair.

The resulting output is a set of partitions that contain all correspondence pairs of entities that we want to use as starting points for the later graph-matching process. There we will decide which entity pairs really match, in our application mainly by utilizing relational similarities. To save space pairs are implicitly stored; they can be reconstructed by building the local cartesian product between two stored identifier lists, while discarded pairs are stored in a blacklist.

The whole process is depicted in figure 1 and the components of the process are further described in the following.

2.1 Preprocess and Block Building

The proposed blocking technique bases on the method presented by Papadakis at al. [8]. Strings are initially split at blanks into tokens. On each token of a split string a domain specific stop word list is applied to omit frequent substrings, that do not contribute to a matching decision and have a high entropy (e.g. *the*). Afterwards the remaining tokens are kept separated (*split-at-blank*-strategy) or the blanks are eliminated and the remaining tokens are fused (*merge-at-blank*-strategy).

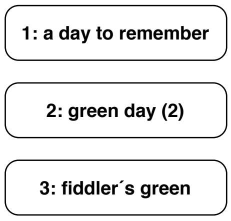

Figure 2: input example

In our approach the incoming entities, which consist of a unique identifier and a string, are initially preprocessed before being handled by the block building process in order to reduce the quantity of noise in the data. Opposite to [8], due to the lack of suitable matching attributes, we do not use a unique identifier accompanied by a set of key-value-pairs, but only a unique identifier and an associated attribute. A set of sample entities can be seen in figure 2.

As previously described, the strings are first separated into tokens using blanks as delimiters. A domain-specific stop word list with common expressions (e.g. *the*, *a* or – in the music domain – *dj*) is applied and afterwards a set of regular expressions is used to eliminate disturbing suffixes like *(1)*, *(3)*, *?*. Consequently some tokens will be removed after this step.

In figure 3 (left) the result of the preprocessing can be seen. For the first entity the stop words *a* and *to* have been removed. The entities 2 and 3 were modified by regular expressions, removing the numerical suffix '(2)' and the apostrophe from entity 3.

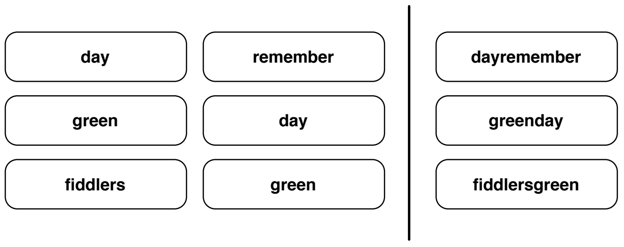

Figure 3: cleansed sample data, (left) split-at-blank, (right) merge-at-blank

In figure 3 the result of both string handling strategies are depicted. On the left side, the remaining tokens are unchanged for the blocking step. The benefit of this approach is that big mistakes can be compensated as we may get a match by other blocks that contain the same entity. On the other hand this may lead to a large number of blocks; entities which consist of more than one token and spelling mistakes will increase the count even further. Additionally, this approach does not cover cases when blanks were omitted and tokens were mistakenly concatenated in the source data. With the second strategy all tokens of one entity will be collapsed to just one string, which results in just one block. While we are able to handle mistakenly concatenated strings big mistakes cannot be compensated by other blocks but only by significantly reducing the required similarity threshold to assume a match.

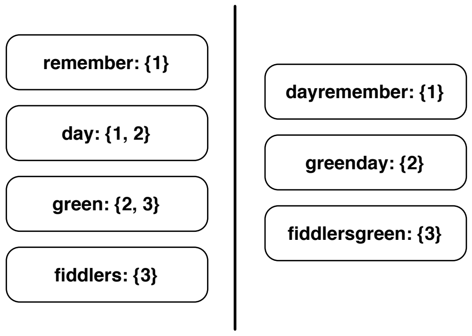

Figure 4: resulting blocks - (left) split-at-blank (right) merge-at-blank

The result of the block building is shown in figure 4. The first strategy places the entities 1 and 2 in the same block for the token *day* and the entities 2 and 3 in the same block for the token *green*. For the desired matching of two data sources it is not required that similar objects of the same source are actually placed in the same block, since a deduplication is usually presumed beforehand.

Each block has a label that indicates the token it originated from and a list of identifiers from entities that contain the token. These labels are used in the block matching, but are irrelevant after the block matching is performed.

2.2 Block matching

After the block building we have to apply a matching between the block sets of both data sources to compute the possible correspondences between entities from both data sources. Since the sources are different we have to handle syntactical heterogeneity, in other words noise. Therefore we must not use exact matching to determine correspondences between blocks from both sources. Because the block labels are stored as strings, we use the Jaro-Winkler string similarity to measure the similarity of block labels, and we need a threshold to determine whether two blocks represent possible entity correspondences or not.

To compute the matching, we have to compare the block labels of both block sets. Comparing them to each other is computationally very expensive, so we choose to sort the blocking labels in alphabetical order and apply a sliding window approach to reduce the number of comparisons. We hereby rely on the assumption that spelling mistakes are more likely to appear in the end part of a word. Then a spelling error will not much affect the sorting position.

Let us assume that the block building has been applied on datasets similar to figure 2, using the *split at blank*-strategy. The built blocks for both data sources are depicted in figure 5. Arrows represent the correspondences between the blocks. The blocks on the left side have resulted from the following artist names: 1) *A Day To Remebmer*, 2) *Green Day* and 3) *Fiddler's Green*; we have noise in the form of two transposed characters in the token *remebmer* ($b \leftrightarrow m$). On the right side, the blocks are built from the artist names 5) *The Offspring*, 3) *Grene Day*, 7) *A Day To Remember* and 2) *Fideler's Green*; here we have more noise.

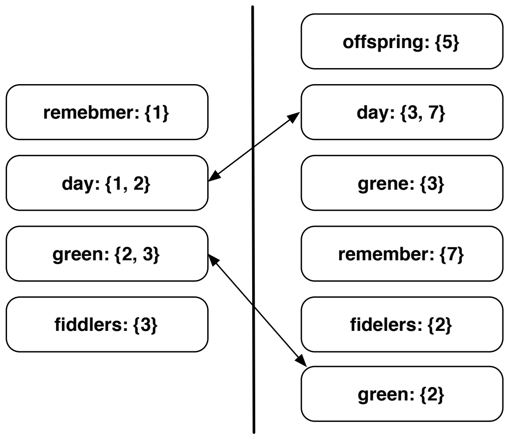

Figure 5: blocking matching result for two block sets using exact matching

If we take the example in figure 5 and assume exact matching, two block matches (*day* and *green*) are considered which would cover a few entity match candidates, like $1-7$ (*A Day To Remebmer*, *A Day to Remember*) or $2-3$ (*Green Day*, *Grene Day*).

In contrast, figure 6 depicts the results for a non-exact matching of the block labels, using the Jaro-Winkler distance; similarities with a threshold of 0.9 or above are considered a block match. With this approach we get an entity match candidate for the artist *Fiddler's Green*, as both tokens have matching partners on each side (*fiddlers* and *fidelers*, *green* and *grene*). With a threshold of 0.9, the artist *The Offspring* does not match with anything, as there is no fitting partner in the left side. Arguably this can change if we would lower the threshold strongly. In this example though, the threshold is low enough to discover all reasonable matchings, but high enough to avoid unreasonable matches.

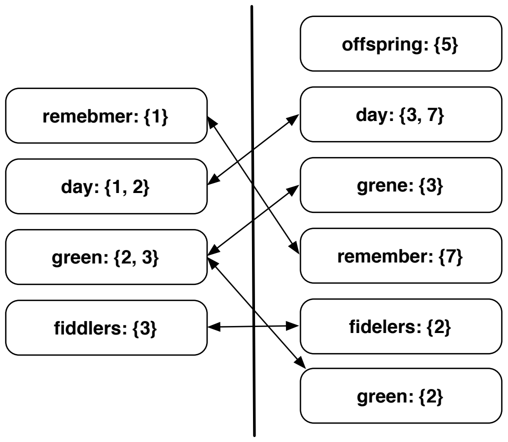

Figure 6: block matching result for two block sets using not-exact matching (threshold: 0.9)

2.3 Partition building

For further computation, we utilize the existing block correspondences as partitions for our later entity matching. Thus we can load the data from all entities that are contained in both blocks in one in-memory step. The partitions consist of two separate lists that each contains the identifiers that are stored in a block. As aforementioned, each partition also will have a blacklist introduced in the next subsection.

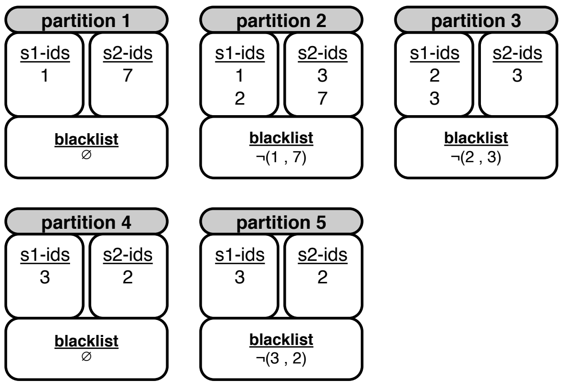

Figure 7: result partitions

As can be seen in figure 7, the partitions are built according to the block matchings from figure 6. Partition 3, e.g. emerges from the matching of the blocks labeled *green* and *grene*. The identifiers are taken from the blocks and added to the identifier lists of the partition. Please note that we do not actually store all pairs to save space; instead we carry two lists along, each containing the identifiers from one source and reconstruct the pairs later.

Additionally two inverse indexes are created alongside creating the partitions (figure 8). The index on each source tells

$1 : p_1, p_2$	$2 : p_4, p_5$
$2 : p_2, p_3$	$3 : p_2, p_3$
$3 : p_3, p_4, p_5$	$7 : p_1, p_2, p_5$

Figure 8: inverse indexes

for each identifier in which partitions it is contained. These indexes will be used in the next step, the pair filtering.

2.4 Pair filtering

Once the partitions are built, we apply a filter to eliminate redundant pairs across the generated partitions. To achieve this, we utilize the inverse indexes that were created in the partition building step and a temporary blacklist, to avoid testing pairs for duplicates more than once. For each partition, we reconstruct all possible pairs using the cartesian product. For each pair, if it is not already contained in the blacklist, we compose two sets by gathering the occurrences in partitions by looking up the inverse indexes. Next, we calculate the intersection of both sets, i.e., all partitions that contain the current pair. We will add the pair to the temporary blacklist and add the pair to the blacklists of all partitions but the first.

The blacklist is later used when the pairs have to be reconstructed using the in-memory cartesian product; blacklisted pairs will be rejected during the reconstruction process.

We can see that three duplicate pairs exist across all five partitions: the pair $(1, 7)$ exists in partitions 1 and 2 and the pair $(2, 3)$ exists in partitions 2 and 3 and the pair $(3, 2)$ exists in partitions 4 and 5. Accordingly the pair $(1, 7)$ is added to the blacklist of partition 2, the pair $(2, 3)$ is added to the blacklist of partition 3 and the pair $(3, 2)$ is added to the blacklist of partition 5. Thereby partition 5 is empty and can consequently be removed.

The partitions, as depicted in figure 7, represent the output of our process and contain the correspondences that we use to initialise a matching process across two graphs representing our data source.

3. PRELIMINARY RESULTS

The blocking technique presented was successfully applied in a project at our institute, namely in the master thesis of Kroll [5] who investigated relational similarity in the context of a matching process for graph databases. We have tested the blocking process with the database dumps of the Discogs and MusicBrainz projects. The Discogs dataset contains over 400 000 artists while the MusicBrainz dataset comprises over 900 000 artists. After the preprocessing and the block building are applied on both datasets, we get different numbers of blocks for the implemented block building strategies. Utilizing the *split-at-blank* strategy, we obtain 242 900 blocks for the Discogs dataset and 435 563 blocks for the MusicBrainz dataset; this corresponds roughly to a reduction by half. With the *merge-at-blank* strategy, we receive 445 763 blocks for Discogs and 903 059 blocks for Musicbrainz; contrary to the other strategy, we see no reduction with regards to the block count.

In our experiments we examined the effectiveness of our pair filtering approach. Table 1 and 2 show the results for both implemented strategies. The aim of these experiments was to investigate the benefit of pair filtering concerning the amount of detected duplicates. For measurement we counted the number of pairs immediately before and after applying the duplicate filtering. All experiments where conducted on a 3.4 GHZ quad-core CPU with 32 Gigabytes of main memory running Linux 4.15 (Debian) and using Java 8u162.

sim	**# before**	**# after**	**duplicates**	**runtime**
			(in %)	(hh:mm:ss)
1.0	734.047.668	728.536.532	0.76	15:31
0.99	745.763.364	739.682.616	0.82	19:28
0.95	810.004.501	799.165.736	1.36	1:05:55
0.9	872.870.493	858.427.029	1.68	3:23:52

Table 1: pair filtering results for split-strategy

In table 1 the results for pair filtering utilizing the *split-at-blank* strategy with various similarities are depicted. With a decreasing similarity threshold we can see that the number of pairs to test increases; yet the amount of duplicates is initially small and does not grow jointly with the number of pairs to be tested for possible duplicates.

When applying the *merge-at-blank* strategy it is obvious that the number of pairs to test is dramatically lower than with the *split-at-blank* strategy. Then we can see that the required runtimes for pair filtering are significantly lower than the corresponding runtimes for the *split-at-blank* strategy, due to the lower pair count. With regards to the amount of duplicates detected by the pair filtering, the results for the *merge-at-blank* strategy in table 2 are similarly underwhelming. In this case, pair filtering yields less than one percent reduction concerning the total amount of pairs.

sim	**# before**	**# after**	**duplicates**	**runtime**
			(in %)	(mm:ss)
1.0	1.873.697	1.867.733	0.32	00:03
0.99	2.013.566	2.004.183	0.47	00:04
0.95	3.607.298	3.574.736	0.91	00:11
0.9	9.185.447	9.103.081	0.90	01:26

Table 2: pair filtering results for merge-strategy

Obviously the chosen attributes, the artist names, of the selected datasets have a high entropy, thus the effect of a post-processing that applies a pair filtering step to eliminate duplicate pairs is ineffective, as the effort spent for the post-processing bears no proportion to the savings. Especially for the *split-at-blank* strategy the increasing runtime with a lowered similarity threshold gets disproportionate.

4. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a new blocking method as part of a process that allows the computation of starting points for the initialisation of a graph matching process. We took steps to adapt the standard blocking technique to datasets that have few attributes that are suitable for a matching. We also addressed the disadvantages of standard blocking by introducing stop word lists and regular expressions to reduce the size and amount of overly large blocks.

Kroll[5] has obtained very satisfactory quality results on the overall graph matching starting with our blocking method. Nevertheless we have to thoroughly evaluate several key aspects of our technique. Foremost we have to evaluate the quality of our approach with regards to the accuracy, especially in comparison to competing techniques.

Additional steps have to be taken for the evaluation of two core elements of our block matching step, the effect of the used similarity function and the optimal size of the sliding window. Concerning the result quality we want to explore our blocking technique with alternatives to the Jaro-Winkler distance and varying sizes of the sliding window and their ramifications on the quality.

5. REFERENCES

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

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

[3] I. P. Fellegi and A. B. Sunter. A theory for record linkage. *American Statistical Association*, 64(328):1183–1210, 1969.

[4] M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. In M. J. Carey and D. A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995.*, pages 127–138. ACM Press, 1995.

[5] H. Kroll. Relationale Ähnlichkeit im Matching-Prozess für Graphdatenbanken. Master's thesis, Leibniz University Hannover, 2017.

[6] A. McCallum, K. Nigam, and L. H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In R. Ramakrishnan, S. J. Stolfo, R. J. Bayardo, and I. Parsa, editors, *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining, Boston, MA, USA, August 20-23, 2000*, pages 169–178. ACM, 2000.

[7] G. Papadakis. *Blocking techniques for efficient entity resolution over large, highly heterogeneous information spaces.* PhD thesis, Leibniz University Hannover, 2013.

[8] G. Papadakis and W. Nejdl. Efficient entity resolution methods for heterogeneous information spaces. In S. Abiteboul, K. Böhm, C. Koch, and K. Tan, editors, *Workshops Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 304–307. IEEE Computer Society, 2011.

[9] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):6, 2006.