Duplicate Table Detection with Xash

Maximilian Koch[1] Mahdi Esmailoghli[2] Sören Auer[3] Ziawasch Abedjan[4]

Abstract: Data lakes are typically lightly curated and as such prone to data quality problems and inconsistencies. In particular, duplicate tables are common in most repositories. The goal of duplicate table detection is to identify those tables that display the same data. Comparing tables is generally quite expensive as the order of rows and columns might differ for otherwise identical tables. In this paper, we explore the application of Xash, a hash function previously proposed for the discovery of multi-column join candidates, for the use case of duplicate table detection. With Xash, it is possible to generate a so-called super key, which serves like a bloom filter and instantly identifies the existence of particular cell values. We show that using Xash it is possible to speed up the duplicate table detection process significantly. In comparison to SimHash and other competing hash functions, Xash results in fewer false positive candidates.

Keywords: data discovery; data lakes; duplicate table detection

1 Introduction

The accelerating decentralized creation and publishing of data as well as the need for integration of such sources has led to a new wave of research on data market places [FSF20] and data lakes [Ar20]. Yet, these centralized data repositories have to deal with the distributed nature of data acquisition and the resulting data quality problems, one of which is duplicate data artifacts. An example of this is the Open Research Knowledge Graph (ORKG) project [Au20; Ja19]. On the ORKG platform, users can categorize and describe contributions from research papers and create additional properties to make them comparable and searchable in a structured form. Out of 524 tabular comparisons in the ORKG dataset, 48 are duplicates (9%). Figure 1 shows an example of two manually created comparison tables from the ORKG sharing duplicate rows [Te22]. As shown in Figure 1, two generated tables on a specific political science topic contain identical content derived from different literature. The discovery of such identical artifacts is useful. However, the attribute labels are rather misleading and the order of rows and columns is different. In a different real-world dataset, the DWTC corpus, containing 174M tables, there are 49M duplicates (28%) [Eb15a; Eb15b].

[1] Leibniz Universität Hannover, Germany koch@dbs.uni-hannover.de
[2] Leibniz Universität Hannover, L3S, Germany esmailoghli@dbs.uni-hannover.de
[3] TIB, Germany auer@tib.eu
[4] Leibniz Universität Hannover, L3S, Germany abedjan@dbs.uni-hannover.de

[5] Screenshots taken from `https://orkg.org/comparison/R110188/` and `https://orkg.org/comparison/R110245/`

doi:10.18420/BTW2023-18

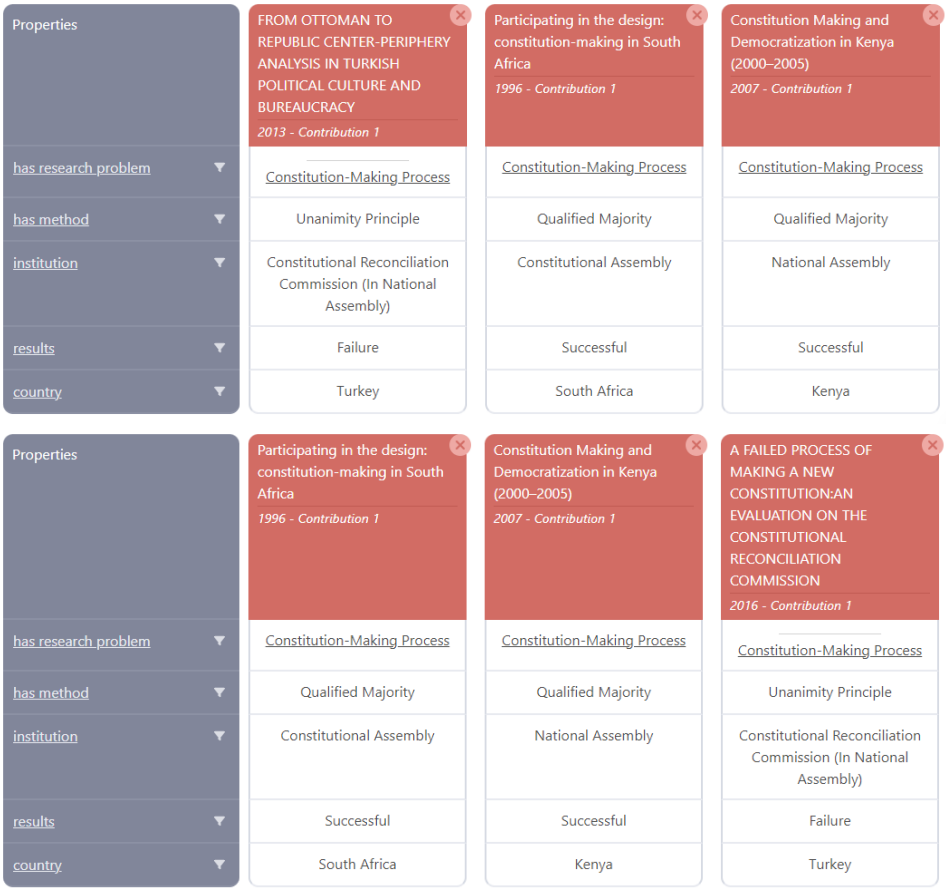

Fig. 1: Duplicate table example in the Open Research Knowledge Graph.[5]

Duplicate detection has been the focus of research for several decades. Most existing work focuses on record linkage, i.e., finding records that represent the same real-world entity [Ch12a; Ch12b; Li20; LSR21; Th20]. Another line of research has dealt with the identification of (near-)duplicate documents [CGS03; Jo72]. In general, the fundamental challenges in duplicate detection are that pairwise comparisons of all considered entities are computationally expensive, i.e., $O(n^2)$ for n entities, and that effective similarity metrics are necessary to capture non-exact matches. To reduce the number of table-to-table comparisons one has to resort to pre-filtering techniques that apriori discard non-matching pairs.

In this paper, we focus on the discovery of *duplicate tables* within a data lake. We consider two modes of duplicate table detection:

1. *Duplicate table retrieval:* Given a user table, the goal is to identify the existence of all tables that match or contain the given table.
2. *Lake de-duplication:* Given a repository of relational tables, the goal is to identify all duplicate tables within a lake.

We consider two tables T_1 and T_2 to be duplicates if a permutation of columns in T_1 exists T_1^p so that T_1^p and T_2 contain the same set of tuples. While this definition simplifies the problem of duplicate table detection by excluding fuzzy matches, there are still runtime bottlenecks. In table retrieval, the initial identification of candidate tables requires the retrieval of tables with matching rows. In lake de-duplication, we require a blocking technique similar to what has been proposed in the duplicate detection literature [Ch12a; Fi15; Ga22]. After which again table-to-table matches have to be considered. As we exclude fuzzy matches, blocking can be as simple as grouping tables based on the number of rows and columns and then hashing them via Simhash into smaller buckets.

The overarching challenge in both detection modes is that in order to verify the match of two tables one has to compare the entire content of both tables after aligning the columns, which is a computationally expensive process when carried out in a naïve manner. A naïve approach to compare two such tables is to first sort the values inside each row alphabetically (horizontally) and then hash the rows into matching buckets. Note that sorting based on column labels might be misleading as duplicate tables might differ in the actual column labels. To circumvent this, the column position is stored for each unsorted row. For m columns and k rows, this results in $O(2 \cdot m \cdot \log(m) \cdot k + 3 \cdot k)$, i.e., sortation of m values of all k rows and the application of a collision-free hash function to match the sorted rows into k different buckets. A final pass is necessary to verify that the original column order is consistent in all matched rows. For large number of tables, this approach will be prohibitively expensive. Furthermore, in a retrieval scenario, where tables are retrieved via an inverted index, one has to first retrieve a set of candidate tables that partially match on a chosen query column.

In this work, we propose hash-based solutions for fast comparison of duplicate candidates as well as fast discovery of candidates from a large data lake. Our proposed solution is inspired by a previously introduced hash function framework MATE [EQA22], which was designed for efficient discovery of multi-column join candidates. MATE leverages a hash function XASH to mask the existence of each row value of a row within a unified bit string and applies a bloom-filter-inspired approach with a so-called *super key* to discard non-joinable candidates. Doing so speeds up the identification of joinable tables by orders of magnitude.

Inspired by the capabilities of XASH, we explored its application for the table de-duplication case, which in a sense translates to joining two tables on all attributes of both tables. When searching for duplicates, the bloom-filter-like structure can be used to rule out non-duplicate rows without the need of reordering the columns and knowing the schema of the tables.

If the hash values of two rows are not equal, the rows are not equal and do not need to be checked in more detail. Otherwise, it could be evidence of a duplicate table relationship.

In this paper, we explore the application of XASH and other similar hash functions for the two detection modes described above: lake de-duplication and duplicate table retrieval. We describe the workflow of each detection process and possible optimizations when dealing with tables. We compare the pruning power to other hash functions and discuss situations where the application of the filtering with XASH is beneficial over direct comparisons.

2 Related Work

Duplicate table detection is related to several lines of research, such as duplicate web page detection, entity resolution, fuzzy joins, and data discovery.

2.1 Duplicate document detection

Duplicate document detection has been vastly studied to enhance the effectiveness of search engines and diversify search results by finding duplicate web pages and dropping these results from the search output [CGS03; He06]. Web pages are mainly comprised of HTML content and are treated as text documents instead of structured data such as tables. Because of this, the tables in web pages are also treated as pure text [CCB02]. These duplicate detection approaches apply feature generation techniques, such as shingling [Br97], sentence extraction [Ku17], and stop word removal, and tokenization [TSP08].

These techniques do not consider the highly structured nature of relational tables with numerical and distinct columns [Br97; TSP08]. Generally, any approach based on the approximation of table content through stop word removal and tokenization can serve the table grouping step and is orthogonal to our proposed techniques. Once a group of candidate tables is retrieved, the filtering with the super key can take place.

2.2 Entity resolution

In entity resolution, the goal is to discover data records that represent the same entity in the real world [Ch12b; Ch21; KTR10; Si22]. Entity resolution methods leverage matching functions that employ similarity metrics to create clusters of table rows that are candidates to be duplicates [Si22]. To reduce the number of pairwise comparisons, entity resolution typically applies blocking methods for fast discovery and exclusion of non-matching pairs [Ch12a; Fi15].

The main difference between entity resolution and duplicate table detection is that the former only focuses on the similarity of the rows [KPN20]. For two tables to be duplicates, all rows

of the two tables should find a pendant duplicate. Furthermore, the schema should be very similar. Entity resolution, cannot directly apply to table-level discovery. It is challenging to systematically decide on two tables being duplicates based on the similarity score of their rows. Therefore, we focus on the case of exact table duplicates where only reordering of rows and columns is allowed, which is already hard enough for a large set of tables.

2.3 Fuzzy Joins

Fuzzy joins, i.e., non-equi or similarity joins, aim to discover the joinable rows from different tables, where the rows have similar keys [WLF11; Yu16]. Xiao et al. aim to discover near-duplicate tables using join discovery, i.e., set similarity search [Xi11]. They leverage the positioning filter to efficiently prune the search space and discover joinable tables for a given table and a join column. However, they only consider single-attribute joins. Thus, joinable tables are not necessarily duplicates. Because the similarity join discovery algorithms only discover the similarity based on one key column per table, these approaches can only serve at the initial candidate retrieval stage.

2.4 Data Discovery

Other data discovery approaches, such as union discovery [Na18] algorithms, also focus on partial similarities. For instance, two tables with a very high unionability score might not even have a single overlapping value, which means that they are not duplicates while they might be unionable.

3 Fundamentals

The core component of our table duplication approach is the so-called *super key*, which is a bit string aggregated from multiple applications of the hash function Xash developed for multi-attribute join table discovery [EQA22]. In this section, we briefly review important characteristics of Xash to motivate its suitability for duplicate table detection and describe the inverted index structure that maintains the generated super keys.

3.1 Xash Design Goals

The design goal for Xash was to hash each row of a table in a way that within a constant operation it is possible to identify whether the row contains a specific value combination or not. Thus, it has the core properties of a bloom filter as it does not lead to any false negatives and is highly effective in differentiating non-equal but similar rows, regardless of the column order.

Furthermore, XASH is designed in a way that the hash of any set of values is masked by the hash of any of its potential supersets. That is why it can serve for arbitrary multi-attribute join keys.

To differentiate row values that are non-identical, XASH captures differentiating features of each value. For each row value, XASH encodes the least frequently occurring characters, the location of these characters, and the length of the value. To further differentiate non-identical rows that by accident end up with the same set of rare characters being encoded. Given a value of length l, XASH shifts the resulting code l bits to the left. Thus for two values to have the same hash, length, rare character distribution and positions must match.

Given a hash string of n bits, XASH divides the bit string into several segments, one segment per eligible character and a remaining segment to encode the length. Typically the hash size must be chosen in a way that each eligible character (space, 0-9, lower-case a-z) can be covered by at least a 1-bit segment. In practice, with a hash size of 128 and larger, the segments can be larger, which makes it possible to encode the relative location of the occurrence of the encoded characters within the corresponding segment.

The remaining bits that equal to the remainder of the division of hash size and a number of eligible characters are used to encode the length of each encoded row value. The encoding of the length calculates the string length modulo the number of available bits for the length segment. Thus only one bit needs to be set to encode the length.

3.2 XASH-Aggregation per row

After generating the XASH for each value of a row, all hash results are aggregated via a bitwise OR operation. The result of this aggregation is a so-called *super key*.

The super key is generated while indexing the corpus and can now be probed like a bloom-filter to check whether a value combination is included or not. Given the aggregated hash value h_c of a candidate value combination C and the super key h_r of a row r , the operation $h_c \vee h_r$ should result in h_r if there is a chance that the candidate value combination is contained. Similar to a bloom-filter, $h_c \vee h_r \neq h_r$ will always correctly identify that the C is not contained in r, however, might be inaccurate if $h_c \vee h_r = h_r$. In the latter case, additional verification is necessary. Experiments and proofs in prior work show that XASH leads to significantly fewer false positives than the state-of-the-art [EQA22].

Example. To illustrate the hash generation using XASH, we use the following example from the World Bank's current GDP dataset [Wo22] as shown in Table 1.

Figure 2 visualizes the process of XASH generation and the merge into the super key. The hash size is 128 bits. To select the least frequent characters, each row value is converted to lower case and only the characters a-z, 0-9, and space are kept. For each row value, the

Tab. 1: Example data: entry from the World Bank's current GDP dataset.

Country Name	Country Code	2020
Europe & Central Asia (excluding high income)	ECA	3222403620453.33

three least frequent characters are selected (marked bold): euro**p**e central asia exclu**d**ing high inco**m**e); **eca**; 3222**4**0**3**6**2**0**4**5333.

In the hash value, there are three bits available per character. If the average location of one of the least frequent characters is in the first third of the row value, the first bit will be set to 1, the second bit if it is in the second third, and the third bit if it is in the third third. For example, **p** in "europe central asia excluding high income" appears in the first third of the entire value. Thus, the first bit of the **p**-segment is set to 1.

With 128 bits and 37 eligible characters, $128 - 37 * 3 = 17$ bits remain for the length segment. To encode the length of the first row value, $41 mod 17 = 7$ (41 is the length of the row value, 17 the segment length) is calculated, which means that the seventh bit in the length segment will be set to 1.

Lastly, all set bits are shifted left by 41 while the overflow is added from the right, ignoring the length segment.

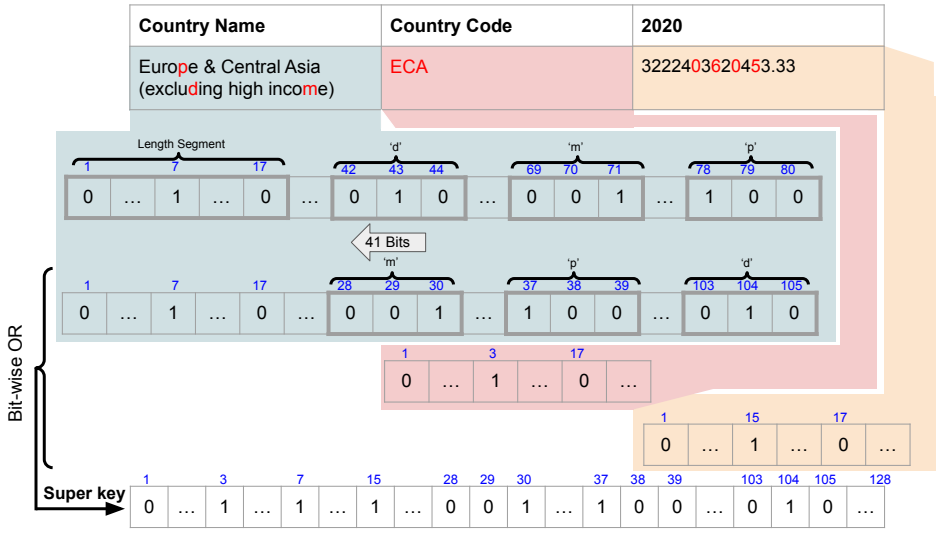

Fig. 2: Example of Xash super key generation

3.3 XASH for duplicate detection

While the probing operation of XASH for join discovery probes for containment, its application for duplicate row detection can be more strict as the containment has to go both ways. Thus, for two rows to be duplicates both hash values have to be exactly the same.

For two tables to be duplicates of each other there has to be a permutation of columns, so that all tuples of one table are contained in the other and vice versa. For practical reasons we relax this duplicate definition as follows:

- We ignore the duplication of rows within a single table and consider two tables to be duplicates as long as each unique row exists in both tables.
- In our implementation, we also enable the discovery of tables that fully contain the rows of another table, meaning that one table has more rows than the other. Note, that the number of columns has to be equal. As we will see in Section 4, this decision does not affect the way XASH is applied for filtering.

3.4 Inverted Index

To achieve fast query results when interacting with the lake, data discovery systems usually leverage one form of an inverted index [Ab16; EQA22; Fe18; Zh19]. An inverted index is well-known in the context of information retrieval and maps tokens to containers, such as documents or tables [GF98]. In the context of data lakes, the containers are the corresponding tables, rows, and columns. As we rely on the MATE framework, we use their inverted index, where an entry inside the index consists of the mapping of the tokenized value to the corresponding table, row, and column IDs [Ab16; EQA22].

In the example shown in Figure 1, the row values 'Turkey', 'South Africa', and 'Kenya' appear in both tables whom for simplicity we assign the ids 1 and 2. With that the content of the inverted index would be as follows:

Turkey $\rightarrow \{1, 2\}$

South Africa $\rightarrow \{1, 2\}$

Kenya $\rightarrow \{1, 2\}$

The schema of the index includes one row of one table from Figure 1 and the super key column is shown in Table 2.

In addition to the aforementioned mapping, the MATE framework also maps the super key to each row value so that information about each row is readily available when probing for one of the row values [EQA22]. Additionally, an index is created on the super key column, to be able to retrieve results from the table when using the super key as a filtering criteria.

Tab. 2: Schema of the inverted index in the database

tokenized	tableid	colid	rowid	super_key
Turkey	1	0	0	...
South Africa	1	1	0	...
Kenya	1	2	0	...

Algorithm 1: Duplicate table retrieval

```
1  Inputs: user_table
2  duplicate_tables = []
3  super_key_mapping = csvToSuperKeyRowIdMap(user_table) /* Map super key to row ids */
4  input_rows = super_key_mapping.values()
5  input_superkeys = super_key_mapping.keys()
6  rows = getDbRowsWithSameSuperKey(input_superkeys) /* Get all rows from the database, that have one of
     the super keys from the input rows */
7  foreach row in rows do
8      input_rows_candidates = super_key_mapping.get(row.superkey) /* Get all rows from the input table, that
         have the same super key as the current db row using hash join */
9      foreach input_candidate in input_rows_candidates do
10         /* check the correspondence of rows, yet make sure that column positions remain consistent across
             matched rows */
11         if verifyRows(input_candidate, row) then
12             table_id_to_rows.add(retrieveTableId(row), retrieveRowId(row));
13             table_id_to_rows_input.add(retrieveTableId(row), retrieveInputId(row));
14 foreach (tableid,rowids) in table_id_to_rows do
15     /* It is checked if there are duplicate tables, based on the detected duplicate rows for each db table */
       duplicate_tables.add(getDuplicateTables(tableid, rowids, table_id_to_rows_input[tableid],
       table_id_to_rows[tableid]));
16 return duplicate_tables;
```

4 Duplicate Table Detection

In this section, we describe how a hash-based index can be used for duplicate table retrieval as an online process for a given user table and as an offline process to de-duplicate a group (a block) of tables. For both applications, we present algorithms that leverage the super keys based on XASH.

4.1 Duplicate Table Retrieval

In the first scenario, the user provides a table and is searching for all possible duplicates or subsuming tables inside the data lake at hand.

Given an inverted index as suggested in Section 3, the naïve approach would use the content of one column to fetch all tables that contain all values of that particular column. Then all remaining columns of the fetched tables would be loaded so that the remaining columns of the input table can be verified against each candidate table.

To reduce the runtime of this step and the number of string comparisons, one can leverage XASH as a filter. This requires us to hash all rows of the input table using XASH and to retrieve the super keys of lake tables that were generated during the indexing phase.

Algorithm 1 depicts the process in more detail. Given the user table, the algorithm first iterates through its rows to calculate the super key for each row and keeps them accessible for later probing.

After processing the input table, the approach retrieves candidate rows from the lake. Using the inverted index, a query is submitted to the data lake where to retrieve any row with a super key that appears in the input table (line 6). Each retrieved row from the data lake is compared with all rows from the input table having the same super key (line 11). This step verifies for two rows with the same super key whether they indeed contain the same row values, regardless of the order.

If an input row and a row from the lake match, the table ID and the row ID of the data lake table as well as the row ID of the input table are temporarily stored in *table_id_to_rows*(line 12) and *table_id_to_rows_input* (line 13).

After all of the rows passed the aforementioned checks, all tables that share any row with the input table are verified (line 15). Using the aforementioned table-to-row maps, tables that either contain all input table rows or are subsets of the input table are identified (line 15). The *getDuplicateTables* function uses the *table_id_to_rows_input* map to check which of the rows of the input table occur in the database table and the *table_id_to_rows* map, to check which rows of the database table occur in the input table.

4.2 Lake De-Duplication

Finding duplicate tables in a data lake requires comparing all pairs of tables.

Typically, some sort of blocking has to be applied to reduce the number of pairwise table comparisons. For the sake of context, Figure 3 shows a conceptual pipeline for Lake De-Duplication including the blocking step. Generally, the blocking strategies have to be tailored to the type of duplicates we are after. If only exact duplicates with arbitrary row and column orders are considered, the blocking can already take the table dimensions into consideration. If fuzzy duplicates are of interest hashing approaches based on Simhash for near-duplicate document detection can be considered. In this paper, we consider the former situation and apply a very simple blocking technique that sorts out tables with equal dimensions. Our approach is orthogonal to blocking. Rather we show, that given a coherent group of tables, i.e., all tables that are in the same block or have the same row and column dimensions and share some general similarity, the existence of the super key significantly improves the overall pairwise comparisons.

All pairs of tables contained in a block must undergo pairwise comparisons. To this end, we join each pair of tables using hash join with XASH super keys and pass the joint table to the validation step. In this step, the joinable rows are validated to discover and drop the false positive rows, i.e., rows that have the same super keys but are not joinable. Ultimately, the tables are duplicates if the number of duplicate rows equals to the number of rows in the smallest table.

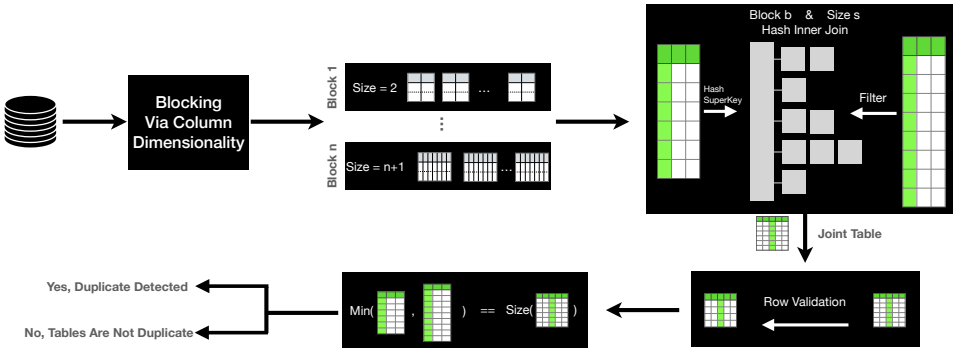

Fig. 3: Lake de-duplication pipeline.

Algorithm 2 shows the process of comparing two tables in detail. *CompareTables* receives two tables as input and returns as the result whether the tables are duplicates, or strictly contained in one direction. First, the smaller and the bigger tables are identified (Line 3). This helps us to create the hash table effectively and to find the candidate subset table. In the first loop, the algorithm (Lines 4-6) creates a hash table based on the super keys in the bigger table. The hash table is created based on the bigger table because in the filtering phase using the smaller table, we can also identify a one-directional subset relationship. Creating the dictionary based on the bigger table allows us to make sure that if even one super key in the smaller table does not exist in the dictionary, we can make sure that the two tables at hand are neither duplicates nor subsets of each other (Lines 9, 10). On the contrary, if a super key from the smaller table exists in the hash table (Line 13), we get the corresponding candidate rows from the larger table (Line 14) and verify them (Line 15). The `verifyRows` compares the actual cell values of rows with the same super key. For this purpose, it sorts the cell values of each row and checks whether the two rows are equal. To save runtime, each row is only sorted at most once and the sorted row is cached in case it is needed for later comparisons. For each matched row the function records the resulting column alignment. If two consecutive row matches result in different column alignments, the tables are considered as non-duplicates. If the rows are duplicates, they will be stored in the *matched_rows* list (Line 16). If the size of *matched_rows* equals the size of the smaller table, it means that the bigger table contains all the rows in the smaller table and there is a subset relationship (Line 17).

The proposed algorithm and pipeline lead to zero false negatives. Based on the definition of duplicates in this paper, two duplicate tables must have the same number of columns.

Algorithm 2: CompareTables

```
Inputs: =t1: table 1, t2: table 2,
matched_rows = []
smaller_table, bigger_table = getSmallerBiggerTable(t1,t2)
foreach row_t1 in bigger_table do
    super_key_t1 = bigger_table[row_t1].get_super_key()
    hashjoin_map[super_key_t1].append(row_t1)
foreach row_t2 in smaller_table do
    super_key_t2 = smaller_table[row_t2].get_super_key()
    if super_key_t2 not in hashjoin_map then
        break
    else
        row_t2_values = smaller_table[row_t2]
        foreach hit_row_t1 in hashjoin_map[super_key_t2] do
            row_t1_values = bigger_table[hit_row_t1]
            if verifyRows(row_t1_values, row_t2_values) then
                matched_rows.append(hit_row_t1, row_t2)
return ||matched_rows|| == ||smaller_table||
```

As our blocking method groups the tables based on their column dimensionality, i.e., the number of columns, there will be no misses in during blocking.

The bloom-filter-like property of Xash assures that any two rows that are duplicates will have the same super key. Therefore, no duplicate rows will be missed if their super keys do not match.

5 Experiments

We carried out a series of experiments to answer the following questions: (1) Can we significantly improve the runtime of duplicate table detection using the Xash filter? (2) How does modifying the Xash generation affect the number of false positives? (3) How do other hash functions perform for the same purpose?

5.1 Experimental Setup

We conduct experiments for both setups: duplicate table discovery and table group de-duplication.

5.1.1 Dataset for duplicate table discovery

To test the efficiency of our approach, we executed duplicate detection tasks on top of the DWTC dataset [Eb15a]. The corpus consists of 145,533,822 tables and is indexed as described in Section 3.

As the algorithm expects a table for input, we randomly selected 5 tables for each row and column number dimensions of 1, 10, 100 and 5, 10, 50, respectively. In all our experiments, we also retrieve tables with subset relationships (see Section 4). Turning this check to exact duplicates does not change the performance.

5.1.2 Dataset for lake de-duplication

To evaluate the effectiveness of the super key filter in a lake de-duplication scenario, we simulate the generation of groups where table-to-table comparisons have to take place at the row level. We sampled coherent groups of tables, i.e., with equal table dimensions, from the Wikipedia dataset[6] [Au07].

The Wikipedia dataset consists of 7,684,431 different tables, with 380,475,701 total entries in the inverted index described in Section 3. For the different test runs on this dataset, we sampled real groups of 1,000-50,000 tables to represent duplicate blocks.

5.1.3 Competitors

For both scenarios, we are interested in the number of false positives and the runtime in comparison to the naïve approach and other hash functions.

- **Naïve approach:** In this approach, duplicate rows are detected by checking if two rows are equal through sortation and step-wise comparison of the aligned row values.
- **Xash:** Two rows are compared by checking if the super keys generated using Xash are equal. The super keys consist of 128 bits, as proposed in the original paper [EQA22]. If the super keys of two rows are equal, the row values need to be checked the same way as described for the naïve approach, to rule out a false positive.
- **SimHash:** SimHash was developed for the purpose of finding similar content using hash values [MJS07]. Its application is similar to using Xash, with the difference that SimHash was used to generate the super keys.
- **CityHash:** CityHash was developed to generate hashes quickly, while still resulting in mostly unique values [PA22]. The application is similar to using Xash or SimHash.

[6] https://databus.dbpedia.org/dbpedia/text/raw-tables

- **MD5:** MD5 was chosen, as it is a broadly used hash function. Its application is similar to using XASH, SimHash, or CityHash.

All experiments were executed on a server with an AMD EPYC 7702P CPU with 64 cores/128 threads, 528 GB RAM, and 10 TB storage space. All code is implemented in Python 3.9.2.

We used PostgreSQL 13.7 to store the inverted index. The indexes map tokenized row values of the dataset tables to the corresponding table, row, and column ids as well as the row super keys. Further, there is an index on the tableids and a third one on the super keys. All code is available on GitHub: `https://github.com/LUH-DBS/XashDedup`.

5.2 Response time in duplicate table discovery

Table 3 displays the results of our main experiments to assess the efficiency of duplicate table retrieval under the presence of different hash functions for super keys. We report the runtime average for five tests per input dimension, i.e., (X rows and Y columns). Furthermore, we report the average percentage of false positives (FP %), which is calculated as $\frac{FPs}{FPs+TPs}$ and measures the ratio of non-duplicate rows that wrongly passed a hash-based filter. In essence, a good filter has a low FP%.

The approach based on XASH has the lowest percentage of false positives across all input table dimensions, except for 10 rows / 50 columns. Note that although for 50 columns the FP rate is about the same rounded number of 100% for CityHash, SimHash and MD5 there are significant differences in the absolute numbers. Using XASH, on average 148k false positives passed the filter, while for other hash functions this number was 750k, 580k, and 632k, for CityHash, SimHash, and MD5 respectively. This difference is clearly reflected in the runtime. XASH yields the lowest average runtime by at least one order of magnitude, compared to the filters with the other hash functions. The experiment is repeated 5 times and the runtime superiority of XASH compared to the other hash functions is statistically significant within 99% confident interval. For 100 rows and 5 columns, the approach based on XASH clearly shows an advantage in terms of false positives and runtime: with only 3.4% of the passed rows being false, the runtime of 688ms is at least 57x faster than the competitors. For the (1,5) dimension SimHash has a slightly better average runtime despite the higher FP rate because in some experiments high similarity of rows inside the input table result in identical hash functions, which in turn reduces the retrieval effort for such input tables.

We make several further observations with regard to the influence of the input dimensions on the overall runtime and FP rate. First, the more columns the input table contains the higher is the overall false positive rate. This is because the hash functions are aggregates of row value hashes. Thus, the more values are hashed and ORed the higher will be the number of 1-bits, i.e., bits that turned to 1, and the more likely it will be that by chance the same

Tab. 3: Runtime, result size, and false positives (FP) **averaged** over 5 experiments per input dimension

	1 Row / 5 Cols		1 Row / 10 Cols		1 Row / 50 Cols	
Average...	Runtime (ms)	FP %	Runtime (ms)	FP %	Runtime (ms)	FP %
XASH	1,023	**1.1**	**320**	**2.3**	**713**	**74.6**
CityHash	34,470	74.6	5,002	100.0	21,611	100.0
SimHash	**963**	**1.1**	19,977	99.9	9,754	100.0
MD5	16,038	50.0	58,353	100.0	21,856	100.0
No Hash	2,639,586	-	1,787,112	-	1,823,862	-
	10 Rows / 5 Cols		10 Rows / 10 Cols		10 Rows / 50 Cols	
Average...	Runtime (ms)	FP %	Runtime (ms)	FP %	Runtime (ms)	FP %
XASH	**135**	**23.4**	**150**	**0.0**	**10,440**	100
CityHash	41,428	99.8	23,420	100.0	36,927	100
SimHash	24,406	100	25,692	100.0	37,610	100
MD5	8,464	98.6	23,238	100.0	370,843	**78.4**
No Hash	403,565	-	2,767,554	-	1,495,390	-
	100 Rows / 5 Cols		100 Rows / 10 Cols			
Average...	Runtime (ms)	FP %	Runtime (ms)	FP %		
XASH	**688**	**3.4**	**397**	**10.5**		
CityHash	39,707	99.9	79,712	100.0		
SimHash	445,152	99.5	71,710	100.0		
MD5	280,832	100.0	80,524	100.0		
No Hash	2,156,850	-	1,454,387	-		

bits are turned to 1. There is an anomaly in the runtime with the (1,5) input datasets for XASH as the runtime is higher than all other chosen dimensions except (10, 50). The reason is that there are too many actual duplicates that need to be processed for (1,5). For larger dimensions, the probability for duplicates and so true positives naturally decreases so that the number of FPs and the rate becomes more relevant. As expected, the approach with no hash function filter displays a significantly higher runtime than the hash-based approaches. We enforced a limit of 1M cells for the number of retrieved rows so that the naïve approach would be able to finish. Small deviations in the number of true positives are a result of a hard limit of the enforced limit. The runtime might be higher in cases where the limit is exceeded, i.e., the other hash functions.

5.3 Runtime Lake De-Duplication

As we make no claims on how to obtain the duplicate groups in lake de-duplication, we only present experiments on the pairwise comparison approach in randomly selected duplicate groups of 1,000 tables (7,284 rows; 54,434 row values, 96 duplicate tables tuples), 5,000 tables (33,937 rows; 233,807 row values, 2,528 duplicate tables tuples), and 10,000 tables (66,091 rows; 404,999 row values, 10,845 duplicate tables tuples). We repeat each experiment five times and report the average.

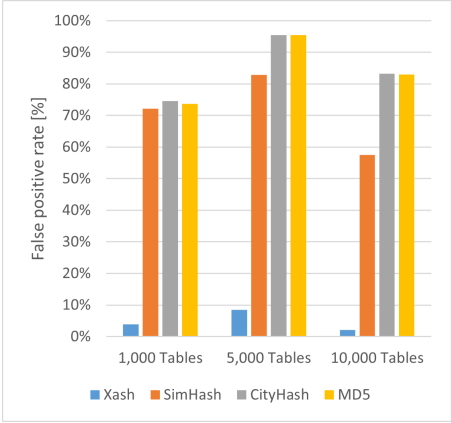

Fig. 4: Avg. false positive rates of table comparisons in duplicate groups

	1,000 Tables	5,000 Tables	10,000 Tables
	Avg. runtime in ms	Avg. runtime in ms	Avg. runtime in ms
Xash	**401**	**9,229**	**48,601**
SimHash	424	9,935	50,920
CityHash	457	10,417	52,610
MD5	458	10,509	52,738
No Hash	7,669	203,769	716,738

Tab. 4: Runtime of table comparisons in duplicate groups

Figure 4 shows the false positive rate and Table 4 contains the runtime for the different hash functions and table groups.

Across all tested groups, Xash has the lowest percentage of false positives with a false positive rate below 2.5%, followed by SimHash with a minimum of 58% false positives, CityHash and MD5 with both around 74% false positives at minimum. Accordingly, Xash application results in the lowest runtime.

The number of false positives increases with the size of the groups for all approaches. Similarly, the runtime of each approach increases each time by an order of magnitude when increasing the group size from 1,000 to 5,000 and then to 10,000.

For 1,000 tables, 2,509,878 row comparisons were performed on average for 5 runs with the naïve comparison algorithm, which leads to a runtime of 7,669ms. Using the MD5 hash function reduces the number of comparisons to 15,797 similar to CityHash with 15,681. Despite the drastic reduction of comparisons by a factor of 160, the runtime is only reduced by a factor of 16, to 458ms for MD5 and 457ms for CityHash. This is due to the fact, that while using a super key eliminates some row comparisons, the super keys still need to be compared with each other using a hash join approach. Furthermore, the overhead of

retrieving tables becomes a more noticeable process. Using XASH, there are on average only 67 false positives out of 1,734 rows, making its approach the fastest. XASH outperforms SimHash significantly within the 99% confidence interval.

For groups of 5,000 and 10,000 tables, we see a similar relative performance relationship as observed for the group of 1,000 tables. Using XASH results in the fastest execution time of 48,601ms for 265,067 rows in 10,000 tables, while SimHash, CityHash, and MD5 are slightly slower. Using SimHash leads to the second-best runtime with 50,920ms, but around 62x more false positives.

Using no hash functions and therefore comparing all rows with each other results in a runtime of 716,738ms and 270,718,502 row comparisons in the largest group. Using any hash function the runtime can be reduced by more than 93%, 92% with XASH.

5.3.1 Varying the Hash Size

The hash functions usually require size parameters that specify the number of bits the returned hash value has. To examine the effect of the hash size, the *compareTables* algorithm is executed with different hash sizes of 64, 128, and 256 for XASH and Cityhash. For SimHash, 64 and 128 bits were used, as there was no implementation for 256 bits available. The experiment was performed on the 10,000 tables group of the Wikipedia dataset.

Tab. 5: Runtime comparison: different hash sizes

	Runtime in ms	FP	SUM (FP+TP)
XASH 64	32,051	5,725	70,562
XASH 128	31,537	5,535	70,354
XASH 256	31,809	4,455	69,266
SimHash 64	35,813	632,478	697,627
SimHash 128	35,058	277,666	342,792
CityHash 64	38,576	1,891,030	1,956,452
CityHash 128	36,653	1,021,649	1,087,033
CityHash 256	36,753	654,260	719,393

Table 5 shows that increasing the hash size reduces the number of false positives, thus leading to a decreased runtime. For XASH, the number of false positives decreases when using 128 instead of 64 bits, but increases slightly with 256 bits. This is because the FP rate is already very low with 128 bits and increasing the hash size increases the runtime of the hash value checks. When using a 128-bit super key, only 5,535 FPs passed the filter compared to more than 275,000 with SimHash and more than 1,000,000 with CityHash. For SimHash and CityHash, the effect is more significant. Increasing the bits from 64 to 128 nearly halves the number of false positives for SimHash and CityHash from around 630,000 to fewer than 275,000 for SimHash and from more than 1,900,000 to around 1,000,000 for CityHash.

This experiment shows that a larger hash size generally leads to better pruning. Increasing the number of bits for the hash value reduces the number of false positives as there are more bits available to encode the row. However, as seen for XASH, there might be a cap on how much pruning can be achieved. It is important to note that increasing the size requires more disk space for storing the hashes. Increasing the hash size for CityHash from 64 bits to 256 bits quadruples the space required for storing the super keys. In particular, the super key with 64-bit hash space requires 11.3 GB and 0.4 GB for DWTC webtables and Wikipedia, respectively. This increases to 45.2 GB and 1.5 GB when using 256 bits. For large data lakes, this increase could mean significant storage space that could be saved by using a more effective hash function, such as XASH.

5.3.2 Modifying the XASH Generation

It is possible to modify the generation of XASH, either by altering the XASH function itself or by changing the input value based on which the hash value gets generated.

Rotation XASH uses a bit rotation routine to differentiate strings with the same rare characters but different lengths. With the rotation, it is expected that more unique hashes will be generated and as such, the number of false positives will be reduced. We test this assumption by comparing XASH with and without rotation on the different samples of the DWTC dataset.

Table 6 shows the number of false positives with and without rotation. Generally, the runtime differences are not statistically significant.

Tab. 6: Number of false positives with/without rotation

	With rotation	Without rotation
1,000 Tables	**156**	164
5,000 Tables	517	**515**
10,000 Tables	**815**	821

As the shifting of the hash values consumes additional resources, the higher resource consumption makes the use of the rotation unjustifiable with the low number of additional false positives when removing the rotation.

To further explore the influence of the rotation step, we also compared the FP-rate for different table sizes. For this purpose, we sampled 383 tables with 20 columns and systematically removed columns so that the actual duplicate tables remained duplicates with just fewer columns. This experiment as well showed that turning off the rotation showed only a minor improvement in the filtering ability.

Input String So far, the super key for each row is generated using XASH by generating the hash value for each row value and then logically OR'ing all hash values of the row values.

A different approach for generating the super key is to concatenate all values of a row and then generate a single hash value for the concatenated string.

This would have the advantage that fewer hashes need to be generated. If the number of false positives using the concatenated input string for the hash generation is lower or equal to when OR'ing the row value hashes, the concatenation method could be preferred as the hash function will have a higher overhead.

To evaluate this theory, we obtained 1,000 tables having 20 columns from the Wikipedia dataset. The row values of each row in all tables are then sorted. After finding duplicates among the tables and recording the false positives, we remove the last column from all tables. The tables that now contain 19 columns are tested for duplicates again and the false positives are reported. This is repeated until the tables contain only 1 column each.

We ran each test twice, one time using the super keys generated by logically OR'ing the hash of the row values of each row and one time using the super key generated by concatenating the row values of each row and then generating the XASH value. The super keys were generated using XASH with 64 bits.

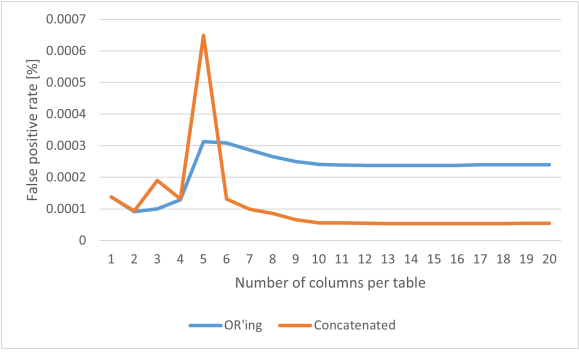

Fig. 5: False positives of XASH with different input formats

Figure 5 shows the FP rate for each group of columns per table for both approaches. It can be observed that generating the super key based on OR'ing has fewer false positives for <6 columns per table, while concatenation has fewer false positives for ≥6 columns per table

The false positive rate is increased when using OR'ing when there are more columns per table. This is due to an increased number of 1-bits existing in the super key when more hash values are combined using the bitwise OR. When using concatenation to generate the super key, this problem does not occur.

This characteristic also explains the spike using concatenation on tables with fewer columns.

When using concatenation, there is still the same limited number of 1-bits is used as for more columns, while OR'ing uses more 1-bits to encode the rows more uniquely.

5.4 Experimental Summary

The number of false positives has a direct influence on the runtime of the pairwise table comparison and duplicate table discovery. The false positive ratio increases with the number of columns as the same hash function has to represent more information. We also conducted preliminary experiments on the effect of Null values on our hash functions. For input tables that only contain null values in one or multiple rows, all hash functions have an equally high percentage of false positives and therefore a high runtime. The same row values produce the same hash value. When the hash values are logically OR'ed, the super key generated is the same for the row, no matter whether the table consists of 1 or 10 columns.

Increasing the hash size decreased the number of false positives for all hash functions. This however leads to more storage space required to store the hashes.

6 Conclusion

We explored the benefits of using hash-based filters in finding duplicate tables. We showcased the duplicate table discovery use case as well as the pairwise comparison use case for lake de-duplication.

The evaluation shows that using hash functions generally improves the overall runtime. In particular, Xash shows the highest promise, followed by SimHash, CityHash, and MD5. In comparison to the original use case of multi-column join discovery, one can say that it is possible to further simplify Xash for efficiency reasons as rotation plays a minor role. The duplicate detection setting is already stricter than join discovery as the probing requires the equality of the hash functions and one-sided containment.

A challenge for all hash functions is when tables with many columns have to be encoded, as the length of a row negatively impacts the pruning power of the hash function.

Future improvements for table de-duplication could be to consider hashing for the grouping phase of large table corpora and to devise algorithms that are independent of lake indexes. Furthermore, it would be interesting to research fuzzy table duplicates. Our current approaches consider tables to be duplicates only when two tables contain the same set of tuples regardless of row and column order.

Acknowledgements. This project has been supported by the German Research Foundation (DFG) under grant agreement 387872445.

References

[Ab16] Abedjan, Z.; Morcos, J.; Ilyas, I. F.; Ouzzani, M.; Papotti, P.; Stonebraker, M.: DataXFormer: A robust transformation discovery system. In: Proceedings of the International Conference on Data Engineering (ICDE). IEEE Computer Society, pp. 1134–1145, 2016, URL: https://doi.org/10.1109/ICDE.2016.7498319.

[Ar20] Armbrust, M.; Das, T.; Paranjpye, S.; Xin, R.; Zhu, S.; Ghodsi, A.; Yavuz, B.; Murthy, M.; Torres, J.; Sun, L.; Boncz, P. A.; Mokhtar, M.; Hovell, H. V.; Ionescu, A.; Luszczak, A.; Switakowski, M.; Ueshin, T.; Li, X.; Szafranski, M.; Senster, P.; Zaharia, M.: Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. Proceedings of the VLDB Endowment (PVLDB)/, pp. 3411–3424, 2020.

[Au07] Auer, S.; Bizer, C.; Kobilarov, G.; Lehmann, J.; Cyganiak, R.; Ives, Z.: DBpedia: A Nucleus for a Web of Open Data. In: The Semantic Web. Springer Berlin Heidelberg, pp. 722–735, 2007.

[Au20] Auer, S.; Oelen, A.; Haris, M.; Stocker, M.; D'Souza, J.; Farfar, K. E.; Vogt, L.; Prinz, M.; Wiens, V.; Jaradeh, M. Y. Bibliothek Forschung und Praxis 44/3, pp. 516–529, 2020, URL: https://doi.org/10.1515/bfp-2020-2042.

[Br97] Broder, A. Z.; Glassman, S. C.; Manasse, M. S.; Zweig, G.: Syntactic Clustering of the Web. Comput. Networks 29/8-13, pp. 1157–1166, 1997, URL: https://doi.org/10.1016/S0169-7552(97)00031-7.

[CCB02] Cooper, J. W.; Coden, A.; Brown, E. W.: Detecting similar documents using salient terms. In: Proceedings of the International Conference on Information and Knowledge Management (CIKM). ACM, pp. 245–251, 2002, URL: https://doi.org/10.1145/584792.584835.

[CGS03] Conrad, J. G.; Guo, X. S.; Schriber, C. P.: Online duplicate document detection. In: Proceedings of the International Conference on Information and Knowledge Management (CIKM). ACM Press, 2003.

[Ch12a] Christen, P.: A Survey of Indexing Techniques for Scalable Record Linkage and Deduplication. IEEE Transactions on Knowledge and Data Engineering (TKDE) 24/9, pp. 1537–1555, 2012.

[Ch12b] Christen, P.: Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection. Springer Publishing Company, Incorporated, 2012, ISBN: 3642311636.

[Ch21] Christophides, V.; Efthymiou, V.; Palpanas, T.; Papadakis, G.; Stefanidis, K.: An Overview of End-to-End Entity Resolution for Big Data. ACM Comput. Surv. 53/6, 127:1–127:42, 2021, URL: https://doi.org/10.1145/3418896.

[Eb15a] Eberius, J.: The Dresden Web Table Corpus, 2015, URL: https://wwwdb.inf.tu-dresden.de/misc/dwtc/, visited on: 04/27/2022.

[Eb15b] Eberius, J.; Thiele, M.; Braunschweig, K.; Lehner, W.: Top-k Entity Augmentation Using Consistent Set Covering. In: Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM). Association for Computing Machinery, 2015.

[EQA22] Esmailoghli, M.; Quiané-Ruiz, J.-A.; Abedjan, Z.: MATE: Multi-Attribute Table Extraction. In: Proceedings of the VLDB Endowment (PVLDB). Apr. 2022.

[Fe18] Fernandez, R. C.; Abedjan, Z.; Koko, F.; Yuan, G.; Madden, S.; Stonebraker, M.: Aurum: A Data Discovery System. In: Proceedings of the International Conference on Data Engineering (ICDE). IEEE Computer Society, pp. 1001–1012, 2018, URL: https://doi.org/10.1109/ICDE.2018.00094.

[Fi15] Fisher, J.; Christen, P.; Wang, Q.; Rahm, E.: A Clustering-Based Framework to Control Block Sizes for Entity Resolution. In. KDD '15, Association for Computing Machinery, Sydney, NSW, Australia, pp. 279–288, 2015, ISBN: 9781450336642, URL: https://doi.org/10.1145/2783258.2783396.

[FSF20] Fernandez, R. C.; Subramaniam, P.; Franklin, M. J.: Data Market Platforms: Trading Data Assets to Solve Data Problems. Proceedings of the VLDB Endowment (PVLDB)/, pp. 1933–1947, 2020.

[Ga22] Gagliardelli, L.; Papadakis, G.; Simonini, G.; Bergamaschi, S.; Palpanas, T.: Generalized Supervised Meta-blocking. Proceedings of the VLDB Endowment (PVLDB) 15/9, pp. 1902–1910, 2022, URL: https://www.vldb.org/pvldb/vol15/p1902-gagliardelli.pdf.

[GF98] Grossman, D. A.; Frieder, O. In: Information Retrieval: Algorithms and Heuristics. Springer US, pp. 134–137, 1998.

[He06] Henzinger, M. R.: Finding near-duplicate web pages: a large-scale evaluation of algorithms. In: Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR). ACM, pp. 284–291, 2006, URL: https://doi.org/10.1145/1148170.1148222.

[Ja19] Jaradeh, M. Y.; Oelen, A.; Farfar, K. E.; Prinz, M.; D'Souza, J.; Kismihók, G.; Stocker, M.; Auer, S.: Open Research Knowledge Graph: Next Generation Infrastructure for Semantic Scholarly Knowledge. In: Proceedings of the 10th International Conference on Knowledge Capture. K-CAP '19, Association for Computing Machinery, Marina Del Rey, CA, USA, pp. 243–246, 2019, ISBN: 9781450370080, URL: https://doi.org/10.1145/3360901.3364435.

[Jo72] Jones, K. S.: A statistical interpretation of term specificity and its application in retrieval. Journal of Documentation 28/1, pp. 11–21, Jan. 1972.

[KPN20] Koumarelas, I. K.; Papenbrock, T.; Naumann, F.: MDedup: Duplicate Detection with Matching Dependencies. Proceedings of the VLDB Endowment (PVLDB) 13/5, pp. 712–725, 2020, URL: http://www.vldb.org/pvldb/vol13/p712-koumarelas.pdf.

[KTR10] Köpcke, H.; Thor, A.; Rahm, E.: Evaluation of entity resolution approaches on real-world match problems. Proceedings of the VLDB Endowment (PVLDB) 3/1, pp. 484–493, 2010, URL: http://www.vldb.org/pvldb/vldb2010/pvldb%5C_vol3/E04.pdf.

[Ku17] Kumar, N.; Antwal, S.; Samarthyam, G.; Jain, S.: Genetic optimized data deduplication for distributed big data storage systems. In: 2017 4th International Conference on Signal Processing, Computing and Control (ISPCC). Sept. 2017.

[Li20] Li, Y.; Li, J.; Suhara, Y.; Doan, A.; Tan, W.: Deep Entity Matching with Pre-Trained Language Models. Proceedings of the VLDB Endowment (PVLDB) 14/1, pp. 50–60, 2020, URL: http://www.vldb.org/pvldb/vol14/p50-li.pdf.

[LSR21] Lerm, S.; Saeedi, A.; Rahm, E.: Extended Affinity Propagation Clustering for Multi-source Entity Resolution. In: Datenbanksysteme für Business, Technologie und Web (BTW 2021), 19. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme"(DBIS), 13.-17. September 2021, Dresden, Germany, Proceedings. Vol. P-311. LNI, Gesellschaft für Informatik, Bonn, pp. 217–236, 2021, URL: https://doi.org/10.18420/btw2021-11.

[MJS07] Manku, G. S.; Jain, A.; Sarma, A. D.: Detecting near-duplicates for web crawling. In: Proceedings of the International World Wide Web Conference (WWW). ACM Press, 2007.

[Na18] Nargesian, F.; Zhu, E.; Pu, K. Q.; Miller, R. J.: Table Union Search on Open Data. Proceedings of the VLDB Endowment (PVLDB) 11/7, pp. 813–825, 2018, URL: http://www.vldb.org/pvldb/vol11/p813-nargesian.pdf.

[PA22] Pike, G.; Alakuijala, J.: Introducing CityHash | Google Open Source Blog, 2022, URL: https://opensource.googleblog.com/2011/04/introducing-cityhash.html, visited on: 08/25/2022.

[Si22] Simonini, G.; Zecchini, L.; Bergamaschi, S.; Naumann, F.: Entity Resolution On-Demand. Proceedings of the VLDB Endowment (PVLDB) 15/7, pp. 1506–1518, 2022, URL: https://www.vldb.org/pvldb/vol15/p1506-simonini.pdf.

[Te22] Technische Informationsbibliothek: Comparisons - ORKG, 2022, URL: https://orkg.org/about/15/Comparisons, visited on: 08/02/2022.

[Th20] Thirumuruganathan, S.; Tang, N.; Ouzzani, M.; Doan, A.: Data Curation with Deep Learning. In: Proceedings of the International Conference on Extending Database Technology (EDBT). OpenProceedings.org, pp. 277–286, 2020, URL: https://doi.org/10.5441/002/edbt.2020.25.

[TSP08] Theobald, M.; Siddharth, J.; Paepcke, A.: SpotSigs: robust and efficient near duplicate detection in large web collections. In: Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR). ACM, pp. 563–570, 2008, URL: https://doi.org/10.1145/1390334.1390431.

[WLF11] Wang, J.; Li, G.; Feng, J.: Fast-join: An efficient method for fuzzy token matching based string similarity join. In: Proceedings of the International Conference on Data Engineering (ICDE). IEEE Computer Society, pp. 458–469, 2011, URL: https://doi.org/10.1109/ICDE.2011.5767865.

[Wo22] World Bank: GDP (current US$), Apr. 2022, URL: https://data.worldbank.org/indicator/NY.GDP.MKTP.CD, visited on: 04/27/2022.

[Xi11] Xiao, C.; Wang, W.; Lin, X.; Yu, J. X.; Wang, G.: Efficient similarity joins for near-duplicate detection. ACM Transactions on Database Systems (TODS) 36/3, pp. 1–41, 2011.

[Yu16] Yu, M.; Li, G.; Deng, D.; Feng, J.: String similarity search and join: a survey. Frontiers Comput. Sci. 10/3, pp. 399–417, 2016, URL: https://doi.org/10.1007/s11704-015-5900-5.

[Zh19] Zhu, E.; Deng, D.; Nargesian, F.; Miller, R. J.: JOSIE: Overlap Set Similarity Search for Finding Joinable Tables in Data Lakes. In: Proceedings of the International Conference on Management of Data (SIGMOD). ACM, pp. 847–864, 2019, URL: https://doi.org/10.1145/3299869.3300065.