MATE: Multi-Attribute Table Extraction

Mahdi Esmailoghli
Leibniz Universität Hannover & L3S Research Center
Hannover, Germany
esmailoghli@dbs.uni-hannover.de

Jorge-Arnulfo Quiané-Ruiz
TU Berlin
Berlin, Germany
jorge.quiane@tu-berlin.de

Ziawasch Abedjan
Leibniz Universität Hannover & L3S Research Center
Hannover, Germany
abedjan@dbs.uni-hannover.de

ABSTRACT

A core operation in data discovery is to find joinable tables for a given table. Real-world tables include both unary and n-ary join keys. However, existing table discovery systems are optimized for unary joins and are ineffective and slow in the existence of n-ary keys. In this paper, we introduce Mate, a table discovery system that leverages a novel hash-based index that enables n-ary join discovery through a space-efficient super key. We design a filtering layer that uses a novel hash, Xash. This hash function encodes the syntactic features of all column values and aggregates them into a super key, which allows the system to efficiently prune tables with non-joinable rows. Our join discovery system is able to prune up to 1000x more false positives and leads to over 60x faster table discovery in comparison to state-of-the-art.

PVLDB Reference Format:
Mahdi Esmailoghli, Jorge-Arnulfo Quiané-Ruiz, and Ziawasch Abedjan. MATE: Multi-Attribute Table Extraction. PVLDB, 15(8): 1684 - 1696, 2022.
doi:10.14778/3529337.3529353

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/LUH-DBS/MATE.

1 INTRODUCTION

There is an increasing interest in enriching existing datasets with additional relevant datasets from large data silos and data lakes. In this context, efficient data discovery systems [15, 28] become essential for many use cases, such as feature extraction [7, 13, 30, 38], data cleaning [43], transformation discovery [1, 32], and data ecosystems [40].

Given a dataset at hand, the most basic requirement of a table to be relevant for enrichment is its joinability [6, 10, 12, 13, 15, 16, 29, 38, 41, 43, 47, 49]. However, existing works on indexing and retrieving joinable tables from large corpora are limited to unary joins, i.e., where a single column is the join key. This limitation restricts the application and the efficiency of discovery systems for prevalent datasets with composite join keys,i.e., join keys composed of multiple columns [18]. For example, up to 37.5% of the primary keys in the TPC-E and TPC-H benchmarks are composite keys. Further, TPC-H and TPC-E benchmarks contain over 168M unique column combinations (UCCs) [17], 99.9% of which are multi-column UCCs. The existence of such combinations is apriori unknown and unexpected for potential discovery tasks. In open data lakes primary key information and other metadata are generally not known. Most keys are often auto-generated columns that are not directly related to the table rows and their corresponding real-world entities. Therefore, a discovery task has to rely on undocumented key candidates. One could discover and index the UCCs, which is an exponentially expensive task w.r.t. both runtime and storage. Considering, our next example, we cannot even claim that join columns in web tables must have the uniqueness property.

Consider the following real-world example, where we explained the root cause of air pollution measured in different European cities[1]. As the sensor data is limited to the three columns (timestamp, location, and pollution ratio) [27], additional dimension tables on weather, public events, and road traffics, were needed to make sense of it. All of the relevant tables had to be discovered and joined based on the timestamp and location columns of the sensor dataset [27]

Using current single-column discovery systems, users have to either repeat the discovery process per key column or check the retrieved joinable tables to remove the false positive (FP) tables and rows, which is a daunting task. According to our benchmark, users might encounter over 1000 times more irrelevant table rows. Thus, one would first find all tables that join with the timestamp and then filter those that have the correct location or vice versa. In either case, there would be an overhead for removing irrelevant tables. For example, searching for joinable tables in the Dresden Web Table Corpus[2]) based on the location and timestamp leads to 700K and 4M candidate tables, respectively. However, only 1, 552 of them contain the exact alignment of time and location that is useful to discover the pollution reasons.

Using a state-of-the-art system, such as JOSIE [47], the runtime for verifying both columns for joins increases by an order of magnitude because discovering tables based on n-ary keys has a factorial time complexity in the number of attributes in each table. Besides, in contrast to the single column key scenario more candidate tables have to be scanned to identify the top-k joinable tables as it is not guaranteed that the joinability of each join column is equally high in each candidate table. Leveraging an inverted index to achieve the efficient discovery of joinable tables is crucial, but building a multi-attribute inverted index requires factorial storage space, which is infeasible.

We propose Mate, a data discovery system to efficiently find n-ary joinable tables from a large corpus with millions of tables for a given query table. To the best of our knowledge, this is the first piece of work addressing the general dataset discovery problem for n-ary key joins at a large scale. Mate employs a pruning technique

Proceedings of the VLDB Endowment, Vol. 15, No. 8 ISSN 2150-8097.
doi:10.14778/3529337.3529353

[1]https://luftdaten.info/
[2]https://wwwdb.inf.tu-dresden.de/misc/dwtc/

to detect the most promising table rows and apriori filter irrelevant tables. It uses a new inverted index element called *super key*, which is an order-independent and fixed-sized hash-value that merges all possible key values into a single index element. This super key allows the system to check the existence of any given key value in the same spirit as a bloom filter. To generate the super key, Mate uses Xash a simple, yet effective, h**ASH** function for cheking the e**X**istance of arbitrary key values. It encodes all cell values based on distinctive properties to avoid collisions of similar join values. As a result, it significantly reduces the FP rate of joinable rows before searching for join attributes and exact matches within a row. In addition to multi-column join discovery, other use cases and application can directly benefit from our approach to beat the exponential dimensionality of multi-column sets. The methods are readily adaptable for duplicate table discovery and union table discovery [3, 4, 8, 23, 30], and spreadsheet transformation joins [48]. For duplicate table detection, our hash function could serve as a prefilter for finding similar records. For table union search, the hash function could be applied in the same spirit as for joins.

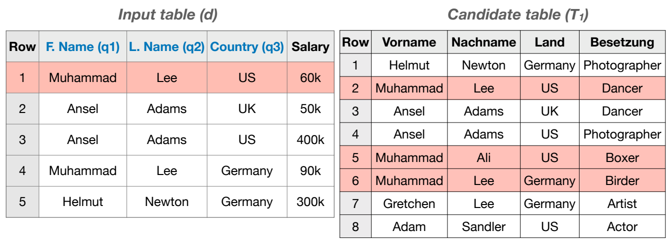
In summary, our major contributions are as follows:

(1) We formulate the general problem of table discovery with n-ary key joins and take the first step in solving this problem with a filter-based approach.
(2) We introduce Xash, which encodes the syntactic attributes of the key values to obtain hash results that optimally use a fixed bit space to avoid overlapping bits of similar join values. The hash function leads effectively filters non-joinable rows and fewer FP rates compared to the state-of-the-art hash functions and bloom filters. Our super key simulates a multi-attribute inverted index with a fixed size hash. We prove that our hash function does not cause any false negatives.
(3) We introduce a two-tier filtering strategy that apriori prunes candidate tables that cannot be part of top-k and candidate rows that are not joinable on all key columns without evaluating the actual row values.

2 PROBLEM STATEMENT

We focus on discovering joinable tables based on composite key joins. Generally speaking, the problem of table discovery is to find the top-k joinable tables for a given query table with a selected composite key [47]. We first formalize the joinability between two tables and then define the problem of n-ary join discovery from large data lakes. We borrow the notations from the literature on inclusion dependencies [9, 33].

Joinability. Intuitively, the joinability between a candidate table (from a corpus) and a given query table represents the corresponding equi-join cardinality. That is, the more key values in a candidate table can be joined with the query table the higher their joinability. Thus, joinability is a measure for the completeness of the join and the relevance of a candidate table to a query table. Formally, assume that $\mathcal{R}$ and $\mathcal{S}$ are two relational schemata and attribute sets X and Y are two subsets of columns where, $X \subseteq \mathcal{R}$ and $Y \subseteq \mathcal{S}$. Without loss of generalization, we can pick any X and Y that consist of the same number of attributes: $|X| = |Y| = m$. In multi-attribute joins $m > 1$. If r is a set of tuples over $\mathcal{R}$, the projection of $\mathcal{R}$ onto X is shown by $\pi_X(\mathcal{R})$, where $\pi_X(\mathcal{R}) = \{t[X] | t \in r\}$. Likewise, s is a set



Input table (d)

Row	F. Name (q1)	L. Name (q2)	Country (q3)	Salary
1	Muhammad	Lee	US	60k
2	Ansel	Adams	UK	50k
3	Ansel	Adams	US	400k
4	Muhammad	Lee	Germany	90k
5	Helmut	Newton	Germany	300k

Candidate table (T_1)

Row	Vorname	Nachname	Land	Besetzung
1	Helmut	Newton	Germany	Photographer
2	Muhammad	Lee	US	Dancer
3	Ansel	Adams	UK	Dancer
4	Ansel	Adams	US	Photographer
5	Muhammad	Ali	US	Boxer
6	Muhammad	Lee	Germany	Birder
7	Gretchen	Lee	Germany	Artist
8	Adam	Sandler	US	Actor

Figure 1: Running example.

of rows over $\mathcal{S}$. We, thus, define the joinability score $\jmath$ between $\mathcal{R}$ and $\mathcal{S}$ on the selected column sets X and Y as:

$$\jmath(\mathcal{R},\ \mathcal{S}) = |\pi_X(\mathcal{R}) \cap \pi_Y(\mathcal{S})|. \tag{1}$$

Yet, calculating Equation 1 is not possible because the one-to-one mapping between the key columns in $\mathcal{R}$ and $\mathcal{S}$ is unknown. Any column permutation of size $|X|$ is a possible candidate. Thus, the joinability between a table with a given composite join key and a table without a defined join key is a factorial number of possible mappings in the number of join key columns. This is why we extend the joinability definition as follows:

$$\jmath(\mathcal{R},\ \mathcal{S}) = \underset{Y'}{\arg\max}\, |\pi_X(\mathcal{R}) \cap \pi_{Y'}(\mathcal{S})|. \tag{2}$$

Here, Y' is a permutation of size $|X|$ from $\mathcal{S}$ where $\jmath(\mathcal{R},\ \mathcal{S})$ is maximum.

Running example. Consider a query table d and a candidate table T_1 as illustrated in Figure 1. Assume the user has selected *F. Name*, *L. Name*, and *Country* as the query columns (columns with blue header). We aim at finding the three columns from table T_1 that result in the highest joinability ($\jmath$) with the query columns from d. If we map *F. Name* from d to *Nachname* from T_1, map *L. Name* to *Vorname*, and map *Country* to *Land*, the one-to-one mapping would lead to a $\jmath$ of 0; If we map *F. Name* from d to *Vorname* from T_1, map *L. Name* to *Nachname*, and map *Country* to *Land*, we would obtain $\jmath = 5$, the maximum joinability score among all possible mappings.

The n-ary join discovery problem. Given a base table d with a relation D, a set of query columns Q, where $Q \subset D$, a corpus of tables T, and a constant value k, the goal is to return the top-k tables from T sorted by their joinability $\jmath$. Solving this problem is challenging for two main reasons:

(1) Calculating the joinability between a given query table and a candidate table with a set of attribute T' requires the mapping between Q and T' that maximizes $\jmath$. The number of possible mappings is calculated as:

$$P(|T'|, |Q|) = \frac{|T'|!}{(|T'| - |Q|)!|Q|!} \tag{3}$$

Here, $P(|T'|,\ |Q|)$ represents the number of possible permutations with size $|Q|$ out of $|T'|$ columns.
(2) Ranking tables based on $\jmath$ and picking the top-k requires calculating the joinability on each candidate table in T.

To discover n-ary joinable tables, one has to (i) build a multi-attribute inverted index that maps different combinations of cell values to their locations in the tables, or (ii) use the inverted index for unary joins and tolerate a large number of FPs. Building a multi-attribute inverted index is not feasible with respect to the storage complexity. For each of the 145M tables inside the Dresden Webtable Corpus, one would need to create $\sum_{i=1}^{c} P(c, i)$ indexes per table. For $c = 5$, the size of the database would increase by more than one order of magnitude. We propose a filtering solution that extends the single-attribute inverted index to obtain both time and space efficiency.

3 PRELIMINARIES

Our approach for enabling multi-attribute joins extends the common inverted index structure.

Inverted index. The inverted index is a structure that maps the content, such as tokens or words, to their containing structures, i.e., tables, rows, and columns [1, 15]. In this work, we extend the index as proposed for the DataXformer system [1]:

$$v_i \mapsto PL_i = \{(T_{i1}, C_{i1}, R_{i1}), (T_{i2}, C_{i2}, R_{i2}), \ldots\}. \quad (4)$$

where, v_i is a value and T_{ij}, C_{ij}, and R_{ij} are the identifiers of the corresponding tables, columns, and rows in the corpus, respectively. This list of triplets is called *Posting List* (PL). We also call every single triplet a PL item.

Discussion. Many state-of-the-art systems leverage the inverted index with small alternations [6, 41, 47]. Given a single-attribute join key, the current systems retrieve the PL items for each value in the key column and the number of returned PL items represents the joinability score for each table. To use the same benefits and optimizations for n-ary joins, we need to generate a multi-attribute inverted index that maps every possible combination of cell values to their location in the tables. For instance, in our running example, we would like to have a PL item that maps the key value of <“Muhammad”, “Lee”, “US”> to the rows that contain both of these values at the same time.

One can use a straightforward algorithm that leverages the original inverted index to find the multi-attribute joinable tables. This algorithm obtains the PLs of one single query column first and then verifies whether the values of the remaining query columns appear in the same tables and rows.

As previously discussed, this approach leads to a large number of false positive rows that require a second verification step. A *false positive row (FP row)* is a row from a candidate table that only contains a subset of join attribute values. For instance, if the search goal is to find joinable tables based on a given 2-column key, candidate rows that are retrieved based on one attribute of the join key and thus only contain one value of the key value combination are considered as FPs and should be excluded from the joinability calculation. A *false positive table (FP table)* is a candidate table with FP rows that is not among the top-k joinable tables. From here on, we refer to FP rows as FPs unless we explicitly specify the type of FPs. The FP rate can be up to 1000 times higher than the actual number of joinable table rows. For each additional row, the discovery system has to compare each value to the composite key values of the input query.

Example 2. Going back to our example in Figure 1, for the value “Muhammad” in the 1st row of the query column q_1, there are three hits in the 2nd, 5th and 6th rows of T_1 and column *Vorname*. These rows are highlighted in red. To find the exact matches for the

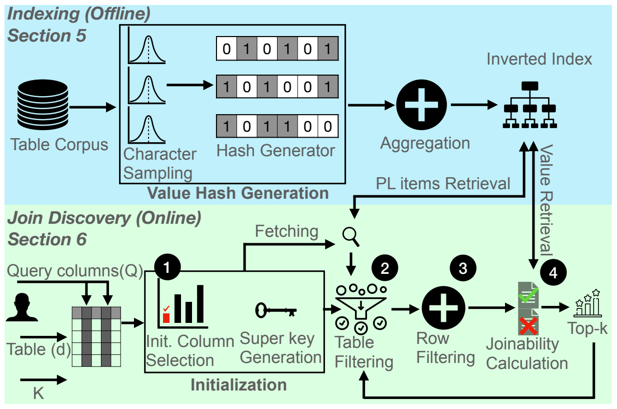


Figure 2: The overall workflow of MATE.

remaining key values “Lee” and “US” in the second and third query columns (q_2 and q_3), the system has to check every other value in rows 2, 5, and 6 in Table T_1. Here, MATE checks 9 value matches only for the first key value <“Muhammad”, “Lee”, “US”>. Alternatively, one would have to send two other independent queries against the index to obtain all rows where “Lee” and “US” occur. However, if an oracle could confirm or deny the existence of “Lee” and “US” with a single operation, the system would only need to check 3 values. This optimization would drastically improve the runtime for calculating the joinability score j for multi-attribute keys. Thus, an index element is required that conveys evidence of whether a particular row contains a given composite key or not. We call this additional element, *super key*.

4 SYSTEM OVERVIEW

Figure 2 depicts the abstract workflow of our proposed system, MATE. In the offline phase, MATE builds an inverted index structure for the tables in the corpus. In the online phase, i.e. discovery phase, MATE uses the inverted index to find the top-k joinable tables among all candidate tables for a given input table.

Indexing step (Offline). To efficiently discover tables that are joinable to a given table, we propose an extension to the state-of-the-art single-value inverted index to efficiently serve the multi-attribute applications. We introduce an additional index element, which is an aggregated hash value called *super key*. The *super key* is space-efficient and does not change the nature of the single-attribute inverted index while serving the purpose of multi-attribute join discovery. The super key is used to prune irrelevant tables and rows to reduce the post-processing overhead. It aggregates the values inside each table-row into a fixed-size entry.

It is worth noting that the super key element does not require any knowledge of the actual keys of a table and serves all possible key combinations inside a table. This way, for a given query dataset and composite key, the system can decide in a single operation if the row is a candidate joinable row or not. Furthermore, the filter does not result in any false negatives. To generate the super key entries, MATE leverages a novel hash function XASH that encodes each row value in a highly disseminative way and aggregates them to the super key. The hash function, as well as the super key, will be discussed in more detail in Section 5.

Discovery step (Online). In the actual data discovery scenario, the user provides a dataset with a composite key and a parameter k, expecting the system to find top-k tables with the highest number of equi-joinable rows to the given input dataset. MATE enables the efficient n-ary key joins by using the super key entry to prune as many irrelevant tables as possible before the joinability calculation. This process undergoes four phases: *(i)* initialization, *(ii)* table filtering, *(iii)* row filtering, and *(iv)* the joinability calculation.

In the initialization step, first, the system selects an initial query column from the composite key Q based on a simple cardinality-based heuristic. The goal of the initial query column is to reduce the number of fetched tables to the minimum by picking the column that leads to the smaller number of PL items from the corpus. That is why MATE selects a single column to fetch the initial set of candidate tables from the corpus. The column selection can be supervised and preempted by the user. Then, the system generates the super key entry for the join columns of the input dataset, i.e., generating a hash-code for each key value combination and aggregating them into a single hash. In the filtering step, MATE applies two levels of pruning for each table: table-level and row-level. With the table-level pruning, MATE decides whether the current table is a promising table to be one of the top-k tables. With the row-level pruning, MATE checks for non-joinable rows in the candidate table. It compares the super keys of the input dataset with those inside the candidate table to drop irrelevant rows from further joinability verification. Finally, MATE retrieves the exact values from the table corpus to compute the final joinability score $\jmath$ for the remainig tables and rows. After calculating the $\jmath$, the set of top-k tables is updated and the next candidate table undergoes the pruning steps. We detail the online phase in Section 6.

5 INDEXING AND SUPER KEY GENERATION

We propose an index structure that retains the space efficiency of the single-attribute inverted index as introduced in Section 3.

5.1 Desiderata

As generating a multi-attribute inverted index requires an exponential number of index entries per table, it is important to extend the traditional single-attribute inverted index to be applicable for n-ary joins. Ideally, we need an additional index element per table row that exposes the existence of join attribute value combinations. We, thus, add an additional element to the index structure named *super key*. This changes the inverted index defined in Equation 4 to $v_i \mapsto \{(T_{i1}, C_{i1}, R_{i1}, S_{i1}), (T_{i2}, C_{i2}, R_{i2}, S_{i2}), \ldots\}$, where S_{ij} is a fixed-size bit array, i.e., *Super Key.*

As one cannot anticipate which attribute combinations are relevant for n-ary joins, we need a hash value that retains them all. The idea is to have a fixed size super key that aggregates hash values for each row value so that we can verify the existence of a composite key without checking all values of the row. MATE aggregates the hash results of individual values into a fixed-sized bit vector using the logical bit-wise `OR` operation. Thus, the super key masks the hash values of each single row value, which ensures that no value is missed, when probing with the super key with the same hash function.

Consider our previous example once again. Rows 2, 5, and 6 are candidate rows to be joined with the first key in the input dataset d based on the first value "Muhammad". Now, to drop the 5^{th} and 6^{th} row from the candidate rows, the super key for these rows should convey that the values "Lee" and "US" do not simultaneously exist in these candidate rows. Yet, the drawback of the aggregated hash value within a fixed hash table is that the super key might mask the hash values of non-existent values as well.

Therefore, the goal is to design a hash function where the aggregation of cell values from different columns results in different super keys. A general approach to this problem is to use a bloom filter. However, off-the-shelf bloom filters have two drawbacks. First, they use hash functions that assume a uniform distribution. Therefore, any arbitrary pair of cell values can result in overlapping bits in the final bit array, which increases the chance of FPs. Second, they are agnostic to the distinguishing properties within columns, i.e., character distributions and positions.

5.2 XASH

We propose XASH, a hash function that encodes syntactic features into distinguishable hashes. As the super key is an OR-aggregation of these hash results, it may mask non-existent values and pass FPs. Thus, our goal is to disperse 1-bits in a way that we decrease the likelihood of two different values from different columns turning the same set of bits to 1. Ideally we want as few 1-bits as possible in the super key to reduce the probability that it covers the super key of random value combinations. XASH leverages three syntactic properties of the cell values to meet this goal: the *least frequent characters*, *their location*, and the *value length.*

5.3 Feature Generation and Encoding

We now turn our attention to how we extract the aforementioned features to apply XASH on a row value. We first discuss the number of bits to generate the hash and its relationship to the table corpus. Then, we explain our segmentation process that allocates different parts of the hash table for different features of a value. Then, we elaborate on the process of mapping the character features to hash bits. Finally, we explain how to use the hash space to relocate the generated bits per cell value to prevent partial matches. We use the example in Figure 3 to elaborate each hash generation step.

5.3.1 Required number of bits. The super key encodes each value into a fixed-size bit array a. On the one hand, as we want the hash values of different individual strings to differ, we want to use all possible bits to encode as many values as possible, i.e., $2^{|a|}$, to reduce the number of collisions. On the other hand, the super key should contain as few '1s' as possible to avoid masking FPs. As a result, underpinning XASH results should be constructed in a way that there is an upper bound of 1 bits for each hash. With this goal in mind, Equation 5 calculates α, i.e., the optimum number of '1s' that is required to generate unique hash values, where C_{unique} is the number of unique values in the corpus and $|a|$ is the hash size.

$$\arg\min_{\alpha} \binom{|a|}{\alpha} > C_{unique}. \tag{5}$$

The binomial term calculates the number of possible bit combinations of size α over $|a|$ bits. The minimum value for α corresponds to the number of '1' bits needed per Xash result. For a 128-bit hash space and $700M$ unique values as existing in DWTC webtables, α is equal to 6. In fact, out of the α bits, one bit is always reserved to encode the value size and $\alpha - 1$ bits for the actual value. Assume that the hash size in our illustrating example is 128 bits and $\alpha = 4$ (3 for the characters and 1 for the value length).

5.3.2 Encoding the characters. As we use $\alpha - 1$ bits to encode each value, we want different values to use different bit segments of a. Thus the characters should be maximally different across words. We can obtain this property based on the character frequency. **Lemma.** Least frequent characters lead to fewer collisions.

Proof. Given a random word w_1 consisting of letters $l_1, ..l_n$ each with a probability of occurrence $P(l_i)$, we sample $K < n$ letters $S = s_1, .., s_k$, where $K = \alpha - 1$. Given a random word w_2 from the same alphabet, we want to reduce the probability to sample the same set of characters K. The probability to obtain a word with the same K characters is $P(s_1) \cdot P(s_2) \cdot \dots \cdot P(s_k)$. This product is minimised whenever a factor is replaced with a smaller probability. Thus, when picking the K least frequent character of the alphabet we obtain $\forall \hat{s}_i \in \{w_1 - S\}, \forall s_i \in S : P(s_i) < P(\hat{s}_i)$. Replacing any s_i with $\hat{s}_i$ results in $P(s_1) \cdot P(s_2) \cdot \dots \cdot P(s_k) < P(s_1) \cdot P(\hat{s}_i) \cdot \dots \cdot P(s_k)$, which leads to a higher probability for a collision. □

To further distinguish the frequencies across domains, we pick the $\alpha - 1$ least frequent characters inside a word as the differentiator. For single-word cell values with flat frequency distributions we draw based on lexicographical order of the characters.
Segmentation. We segment the hash space into smaller blocks, depending on the length of the hash array to encode the features of a cell value. This segmentation specifies how many bits each feature needs. Depending on the number of possible characters, Xash splits the hash array into as many smaller fixed-size segments, one for each character. In our case, we consider all 37 alphanumeric characters including space, which results in 37 segments of size β. We create an additional segment to encode the length of the string value. Sticking with 37 as the number of characters, we can calculate β as follows:

$$\arg\max_{\beta} (37 \cdot \beta < |a|) \tag{6}$$

$|a|$ is the length of the hash array: we have $\beta = 3$ with a hash size of *128* bits. This segmentation dedicates the largest possible subarray to encode character features, because the character features, including the position of the characters, are more discriminative than the length of the value. The rest of the hash array can be allocated to the length segment: $|a_l| = |a| - (37 \cdot \beta)$. For a hash size of 128 bits, the length segment would comprise of 17 bits $(128 - 37 \cdot 3)$. 99.99% of the English words have fewer than 17 characters [26]. Therefore, the explained segmentation can cover almost all possible words in English language. In our real-world data lakes, Dresden webtable and German open data, over 83% of the cell values have at most 17 characters. For larger hash sizes, i.e., 512, $|a_l| = 31$ and covers the length of more than 98% of the cell values in our corpora.

In Figure 3, we want to obtain the hash results for values "muhammad" (C_1), "lee" (C_2), and "us" (C_3) using Xash and then aggregate them into the super key of the row. The red characters in the table cells represent the selected least-frequent characters.

5.3.3 Encoding the character locations. Generally, one bit per character would be enough to encode its occurrence in a value.However, if more bits are available ($\beta > 1$), we can use them to encode the relative character position inside the original string to further distinguish the hash results of different values. For this purpose, we divide the string in β equal areas from left to right. We encode a character by checking in which area the character appears and then we set only the corresponding bit among the β character bits from left to right. More formally, if l_v is the length of the value, i.e., the number of characters in the value, and λ is the location of the character we would like to encode (we take average of the locations in case of character repetition), then, x, i.e., the bit index in the character segment that represents the relative location of the character is calculated as: $x = \left\lceil \frac{\lambda \cdot \beta}{l_v} \right\rceil$

Returning to our illustrating example, we calculate the average location in the original value for each of the least frequent characters. For instance, the average location of the characters "u" and "d" in cell C_1 are 1 and 8 respectively. The first hash array represents the Xash result of C_1. With 3 bits for each character ($\beta = 3$): the characters with the average location of less than 3 ($\frac{8}{3} = 3$) turn the first bit (100); between and including 3 and 5 turn the second bit (010); and above 5 are encoded with the third bit of their corresponding segments (001). For example, characters "u" and "d" in $C1$ are located in the first and the third bits of their segments respectively. We use blue color to trace the character "u" in the hash array.

5.3.4 Encoding the length. With the segmentation step of Xash, a fixed-size segment is dedicated to the length of the value l_v. Storing the actual binary representation of the value length can be problematic as it leads to an unknown number of 1 bits in the segment. This can also mask the length values of the columns with shorter values. For instance, the encoding of $l_v = 7$,i.e., $< 111 >$, can mask the encoding of a value with $l_v = 2$, i.e., $< 010 >$. To address this problem, we reserve one bit per possible $l_v \mod |a_l|$. This way, we maintain the number of used bits i and different length values do not mask each other because each length turns a distinguished bit to set. Encoding the length feature of the values introduces another benefit to the system: Positioning the length segment as the left-most segment allows the system to use short circuit optimization, which skips unnecessary bit operations. If there is no value in the candidate row with the same length as a query value, Mate does not need to check the character segments. Consider our running example. The cell values "Boxer" and "Birder" in rows 5 and 6 respectively. Both of these values start with the encoded character "B". Therefore, "B" and its position cannot differentiate these two cell values. However, the values have different lengths, which makes their hashes distinguishable.

5.3.5 Bit rotation. As a final measure to differentiate hash values without using more 1 bits than i, we reduce the likelihood of so-called random matches through rotation of the character segments. A random match occurs when a key value that is partially masked by the individual hash functions of individual row values is masked by the aggregated super key. For example, we could have a query key that contains rare characters that occur in two different columns. Or

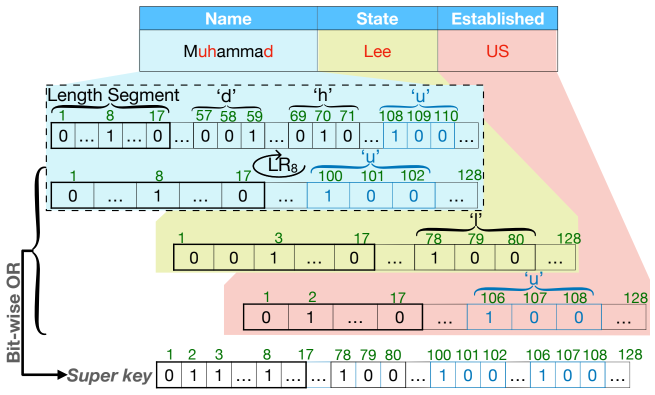


Figure 3: Example of Xash and the results aggregation.

if one value shares the length of the key and the other only shares the characters of the key value that we are searching for. To prevent these kinds of FPs, Xash rotates the character hash segments of a value v by its length l_v to left. The most left bits that fall off will be moved to the beginning of the bit vector. For example, a 3-bit rotation of '01100101' equals '00101011'. Note that the rotation only applies to the character-related segments. As a result, a random match becomes less likely because the length of the random key value must match the length of the column that overlaps in terms of the least frequent characters.

Lemma. Rotation reduces inter-column bit collisions.

Proof. Given that two cell values c_1 and c_2 are masked by a given super key sk, there are two general cases that lead to sk masking false positives. In both cases the length bits must overlap. Either two values from the same domain or from different domains have the same length respectively. In all collision cases, the K characters of each cell value c_1 and c_2 are masked, where $K = \alpha - 1$. Now assume that each of the K characters is rotated based on the length of c_1 and c_2 respectively. Now, for a join candidate $\hat{c}_1$ and $\hat{c}_2$, the collision occurs only if there is an exact match of characters and length between any pair of c_i and $\hat{c}_j$. In all other cases there is a 1-bit in a position that is not masked by sk. We now prove that the opposite leads to a contradiction. There is a collision under rotation of two cell values with different length where there is no exact match between any pair of c_i and $\hat{c}_j$. It follows that $\forall c_i \exists \hat{c}_j : |c_i| = |\hat{c}_j|$ with an injective mapping. Thus, there is a pair of c_i and $\hat{c}_j$ with the same length, where one of the encoded K characters is different. Therefore $K - 1$ character bits will be rotated the same way and are covered. Every other c_i covers at most K characters. Thus, there must be an exact match between c_i and $\hat{c}_j$. □

The previous lemma shows that rotation reduces the number of collisions. This is also shown in our micro-benchmarks where we analyze its impact on filtering. Continuing our illustrating example, the character segment bits all rotate by 8 positions to the left. As can be seen in the rotated array, the segment for "u" moved from the 108th to the 100th most left bit. The final aggregated results for C2 and C3 are shown in separate hash array excerpts. Note that because of the rotation in the hash results, the character "u" appears in different locations for C_1 and C_2 respectively.

Algorithm 1: Mate discovery algorithm.

```
1  Inputs: d: query table, Q: set of query columns, k: the maximum number of
       desired tables
2  TOPK ← empty heap
3  q'← init_column_selection(d, Q)
4  list_of_all_PLs ← fetch_PLs(q')
5  candidate_tables ← extractTable_sorted_ids(list_of_all_PLs)
6  superkey_map_Q← map(Q, generate_super_keys(Q, d))
7  for t in candidate_tables do
8      table_PLs ← extractPLs(t, list_of_all_PLs)
9      if TOPK==k AND size(table_PLs) < TOPK.min().J then
10         BREAK and GOTO Line 23;
11     $r_{checked}$ ← 0
12     $r_{match}$ ← []
13     for pl_item in table_PLs do
14         if TOPK==k AND size(table_PLs) - $r_{checked}$ + size($r_{match}$) <
             TOPK.min().J then
15             Break and GOTO Line 7;
16         d_rows ← superkey_map_Q.get(pl_item.value)
17         for d_row in d_rows do
18             if d_row.superkey ∨ pl_item.superkey = pl_item.superkey then
19                 candidate_rows.add(pl_item)
20         checked_values += 1
21     J ← calculateJ(candidate_rows)
22     TOPK.update(table_id, J)
23 return TOPK
```

5.4 Index Updates

There are three possible types of edits on table corpora that lead to updates in the index-level: *insert*, *update*, and *delete*. Inserting a new table to the corpus requires the following updates. Other than generating the PL items for the cell values in the newly added table, a super key is generated for each row. Inserting a new row to an existing table also follows the same procedure. Adding a new column to an existing table requires applying Xash on each individual column value and replacing the corresponding super key by the result of a bit-wise OR operation with the new Xash result. Updating the value of a cell requires a complete re-hash of the corresponding super key. Deleting a table/row only requires deleting the PL items for the table/row. Yet, deleting a column from a table might change the super key entries, triggering a rehashing of all rows. Although some of the aforementioned updates require regeneration of the super key, the system can handle the changes locally to the affected table.

6 JOINABLE TABLES DISCOVERY

We now discuss how we leverage our index structure and Xash to efficiently find n-ary joins. Recall the discovery phase is comprised of four main steps (see Figure 2): initialization, table filtering, row filtering, and joinability calculation. Algorithm 1 shows the pseudocode of this discovery process. Overall, Mate receives a query table d, a set of columns from the query table as the composite key Q, and an integer k (Line 1) with the goal of discovering the top-k joinable tables for d and based on Q.

In the *initialization* step, Mate selects the initial column, fetches the candidate tables, and applies our hash procedure, i.e., super key generation, on the join columns of the input table (Lines 4-6). In the *table filtering* step, it leverages the number of corresponding index hits per table to prune the tables that cannot be among the top joinable tables (Lines 9-15). We introduce two coarse-grained pruning rules to filter non-promising tables while retaining all relevant tables. In the *row filtering* step, the system prunes as many non-joinable rows as possible for each candidate table (Lines 16-20). It evaluates the rows of candidate tables that are likely to be among the top-k tables, by only checking the membership of the key values through the generated super keys. We detail the aforementioned three steps in the following subsections.

In the final *calculateJ* step, MATE validates the rows that remain after the previous two filtering steps and calculates the joinability score ($\jmath$) of the remaining tables as defined in Section 2. To do so, it fetches the actual cell values of each row, compares them to the composite key values (Line 21). It then updates the list of the top-k joinable tables (Line 22). MATE terminates and reports the discovered joinable tables (Line 23), once it has discovered the top-k joinable tables or checked all the candidate tables.

6.1 Initialization Step

MATE has to initiate the retrieval with one of the key columns in Q using the single column index. The initial column can heavily impact the runtime of the system. Consider the Kaggle IMDB dataset[3], which contains information about more than 5000 movies. If we use it as the query table with the join key <"Director name", "Movie title">, we obtain 500k tables when fetching with the column "Director, and 38M tables when using "Movie Title", respectively.

Yet, selecting the ideal initial query column depends on how many PL items each cell value will match and cannot be known upfront. We, thus, resort to a heuristic that only considers the query table at hand. We choose the column with the smallest number of unique values, i.e., lower cardinality, hoping to match fewer PLs in the index. Thus, `init_column_selection()` picks the column with the minimum cardinality, $q' \in Q$ (Line 3). With the initial column, MATE fetches PL items including their generated super key for the corresponding rows (Line 4). Then, the fetched PL items are grouped based on the table identifiers (Line 5). At the same time, it sorts the tables based on the number of PL items in each table to evaluate more promising tables first and enabling table pruning techniques. Next, MATE generates the relevant super keys for the join columns of the query table. For each column $q \in Q$, it hashes the column values and uses an `OR` operation to generate the aggregated hash for each key value combination (Line 6). We generate a dictionary to map initial query column values to the generated super keys. This helps us to efficiently align fetched super keys with query table super keys in later steps.

6.2 Table Filtering

We propose a *coarse-grained* table filtering approach, inspired by prefix-filtering [6], to further reduce the set of candidate tables. While traversing over the fetched tables, MATE prunes tables that do not stand a chance to be selected as the top-k joinable candidates. For each candidate table (Line 7), it leverages the corresponding PL items of the table (Line 8). Table filtering applies two strategies:
(1) If $\jmath_k$ is the joinability score of the worst table in the top-k list, i.e., table with the lowest joinability score, and L_t is the number of PL items for current candidate table t, the system drops the candidate table t from further evaluations iff $L_t \leq \jmath_k$ (Line 9). As tables are sorted in decreasing order based on their number of PL items, the remaining tables contain equal or smaller number of PL items and cannot end up among the top-k tables. Thus, the discovery task halts when this rule activates for one table (Line 10). Going back to our running example, assume that we aim to discover the top one joinable table to d and we find T_1 as a joinable table with joinability of $\jmath_k = 5$. The moment that a candidate table arrives for evaluation and it only contains 4 or less PL items, the system can return T_1 as the most joinable table. This is because the new and the following candidate tables in the best case, where all the rows are joinable, will have the joinability of 4 and cannot replace T_1.
(2) If r_{checked} is the number of already evaluated rows from a table t and r_{match} is the number of joinable rows among the evaluated rows, MATE drops the current table iff $L_t - r_{\text{checked}} + r_{\text{match}} \leq \jmath_k$ (Line 14). This rule skips the current candidate table, because even if all the remaining rows are joinable to the input table, it cannot achieve a higher joinability score than the worst top-k table. Already during this check, the system leverages the super key to filter joinable rows. Once a table is skipped, the algorithm evaluates the next table (Line 15). Both table filtering strategies only apply after MATE already saw k joinable tables. Once again, assume the previous example, where the user wants to find the top one joinable table to d and T_1 with $\jmath_k = 5$ is already discovered. Previous rule cannot drop a candidate table with 10 PL items. Now, assume that MATE evaluates the first 7 PL items of this newcomer table (See Row Filtering in 6.3) and only one of them are in fact joinable to d. In this case, the system can ignore the remaining PL items because even if the remaining three rows are joinable, the candidate table only reaches the joinability of 4 that does not replace T_1.

6.3 Row Filtering

If there is a prospect of a candidate table to be one of the top-k tables, meaning that the candidate table incorporates enough candidate joinable rows, MATE evaluates them. It leverages the super keys generated for both rows in the candidate and query tables to efficiently check the membership of the key values in the candidate rows. If the super key of a query table row is not masked by the super key from the candidate joinable table, the system can simply drop the table row from joinability computations.

For each candidate row, represented by the PL item (Line 13), MATE discovers which query rows should be compared with the current candidate row. To find the query rows efficiently, the system uses the generated dictionary that maps the initial column values to the corresponding super keys of the composite key value combination (Line 16). In our running example, row 1 in T_1 is a candidate row and MATE binds it to the 5^{th} row in d using a dictionary that maps the cell values to the input rows. This step ensures that we do not iterate over all query table rows for each candidate row. Then, for each corresponding input row `d_row` (Line 17), the super keys of the current candidate row and its corresponding input rows are compared through bit-wise `OR` masking (Line 18). An input row is a join candidate to its corresponding row of the PL item iff the result of the bit-wise `OR` operation on their super keys is equal with the super key of the PL item (Line 19).
Example 3. Consider our running example once again. If the hash values for "Muhammad", "Lee", "US", "Ali", "Germany", "Dancer",

[3]https://www.kaggle.com/carolzhangdc/imdb-5000-movie-dataset

"Boxer", and "Birder" are 1001000, 01100000, 00010100, 00010001, 10001001, 10000010, 10000001, and 00001001 respectively, the aggregated super key for the rows with value "Muhammad", i.e., rows 2, 5, and 6 in the Table T_1, would be 11111110, 11011101 and 11101001 respectively. Likewise the super keys over the columns in Q and for the first row in d would be $1001000 \vee 01100000 \vee 00010100 = 1111100$. In our example, the resulting super key 1111100 is not a subset of 11011101 and 11101001, which are the super keys of rows 5 and 6. Therefore, 5^{th} and 6^{th} rows are removed from further $\jmath$ calculation. On the other hand, 1111100 is subsumed by 11111110, i.e., the super key of the second row of T_1. Therefore, we add the second row to the list of candidate rows. Similar to bloom filters, this operation can lead to FPs but never to false negatives (FNs).

Lemma. The super key will never miss a joinable row.

Proof by contradiction. Given a table R with a row r where c_1, c_2,... are the cell values of the row r. Now assume that we have a deterministic hash function h so that $h(c_1) = h_1, h(c_2) = h_2, \ldots$. The super key sk is constructed as follows: $sk = h_1 \vee h_2 \vee \ldots \vee h_n$. Now assume, there is a table T with a 2-column join key and a row with a value combination $k1, k2$ with $k1 = c1$ and $k2 = c2$, so that the columns are joinable with r but the bitwise OR operation of $h(k_1) \vee h(h_2)$ will not be covered by sk, i.e., $h(k_1) \vee h(k_2) \vee sk \neq sk$

However by construction $h_1 \vee h_1 \vee sk = sk$. It follows $h_1 \neq h(k_1) \vee h_2 \neq h(k_2)$, which is a contradiction to $k1 = c1$ and $k2 = c2$. □

6.4 Analysis

There are two main features of Xash that lead to higher efficiency over BF: 1) The non-uniform distribution of hash values; 2) the encoding of the length. First, unlike BF that is built on the premise of uniform distribution of its underlying hash functions, Xash exploits the knowledge that different values come from different columns. Building on the premise that each domain has unique syntactic features [44], Xash maps hash results to unique segments based on their domain to prevent random overlaps. Assume an $|a|$-bit hash array. The probability of a collision of two words using LHBF [19] is $\frac{1}{\binom{|a|}{2}} = \frac{2}{|a| \cdot (|a|-1)}$. Now consider Xash. To generate a collision one has to hash a value with the same set of K rare characters and the same length, where $K = \alpha - 1$. The probability of drawing the same K characters out of 37 in the same set of β positions is $\frac{1}{37 \cdot \beta} \cdot \frac{1}{36 \cdot \beta} \cdot \ldots \cdot \frac{1}{(37-K+1) \cdot \beta}$. We now solve the inequality $\frac{2}{|a| \cdot (|a|-1)} > \frac{1}{\beta} \cdot \prod_{i=1}^{K} \frac{1}{37-K+1}$. For $|a| = 128$ ($\beta = 3$), the inequality is true for $K > 3$. If we now also include the information about length encoded in the remaining 17 bits we obtain $\frac{1}{|a| \cdot (|a|-1)} > \frac{1}{17} \cdot \frac{1}{3} \cdot \prod_{i=1}^{K} \frac{1}{37-K+1}$, which is already true for $K > 2$. Thus, our approach that explicitly leverages character positions and the length of values leads to fewer collisions when encoding K random characters. Note that the rarity of the characters was not included in this setup.

7 EXPERIMENTS

We evaluate Mate having the following questions in mind: (i) *What is the runtime benefit of using Mate compared to the state-of-the-art inverted index [1] when dealing with n-ary joins?* (ii) *How do parameters, such as the cardinality of tables, the choice of k , $|Q|$, and the hash size affect the runtime?* (iii) *What is the filtering power of Xash compared to other hash functions?*

Table 1: Input query tables.

Query Set	*# of tables*	*Corpus*	*Cardinality*	*Joinability*
WT (10)	150	DWTC	3	4
WT (100)	150	DWTC	16	52
WT (1000)	150	DWTC	151	99
OD (100)	150	German Open Data	15	40
OD (1000)	150	German Open Data	263	1,434
OD (10000)	150	German Open Data	2455	8,187
Kaggle	11	DWTC	34,400	2,318
School	10	School corpus	3,100	15,130

7.1 Experimental Setup

We probed different query tables against two different table corpora: the openly available Dresden Web Table Corpus (DWTC) and the German Open Data repository(https://www.govdata.de/). These corpora are selected because they yield tables with different sizes and dimensionalities. The DWTC corpus contains more than $145M$ tables, $870M$ columns, $1.45B$ rows, and $660M$ unique values. The German open data also contains $17k$ tables, $440k$ columns, $62M$ rows, and $20M$ unique values. As query tables, we leverage sets of random inputs. Similar to the state-of-the-art [47], we selected 900 query tables as six groups (150 tables in each category) from open data and webtable corpora. Each group is randomly selected from different cardinality distributions and we make sure that no duplicate tables are included inside those groups.The query columns are selected randomly. For open data, the groups reflect query tables with cardinality smaller than 100, 1000, and $10k$, respectively. As WTDC tends to have smaller cardinalities, the groups contain tables where the cardinalities are smaller than 10, 100, and 1000. In the experiments, we label the input query table collections as *Corpus (size)*, e.g., *WT (10)* and *OD (100)*. In addition, we also leverage *School* corpus, generated in a previous work [7]. This corpus contains 335 tables. The tables are comprised of over 27 and 30,000 columns and rows on average respectively. This corpus enables us to evaluate the systems in the existence of larger tables. In this set of experiments, we extend the tables containing "Program Type" and "School Name" columns with joinable tables. We also selected machine learning datasets that require enrichment from Kaggle. This query set has more general content, therefore, they run against the webtable corpus. These query tables lead to almost $7M$ joinable candidates on average based on their first query column. Table 1 shows the query table sets used in this paper including statistics about the number of tables, average cardinality, and the average joinability score. In all our experiments, unless specified otherwise, we leverage the 128-bit hash space and we look for top-10 joinable tables. We implemented Mate in *Python 3.7* and ran it on a server with 128 CPU cores, 500 GBs of RAM, and 2 TBs of SSD drive. We used Vertica *v10.1.1-0* [22] to store the index. Our code and datasets are publicly available [4].

Index generation. All of the hash-based solutions including Mate +Xash require a prior hash generation. The original webtable and open data corpus sizes are approximately 250 and 12 GB each with the inverted index SCR. Mate requires additional space to store the super keys. As we defined the super keys per cell value (See Section 5.1), Mate needs $8.3B \times 128b = 123.6$ and $62M \times 128b = 11.9$ Gigabyte for webtables and open data, respectively. Leveraging more efficient structure, where the super keys are stored per row, one can

[4]https://github.com/LUH-DBS/MATE

reduce the index size to $1.45B \times 128b = 21.6$ and $62M \times 128b = 0.92$ Gigabyte, respectively. This space efficient structure might lead to integration overhead, where the super keys are joined with the PLs. Josie on the other hand, requires 293 and 20 Gigabytes additional storage for webtables and open data, respectively. Note that the index for Josie is not sufficient for multi-column join discovery as it does not store the rows information of the cell values. Therefore, Josie-based approaches require an inverted index like SCR. The offline index generation for MATE takes 35 and 2 hours for webtable and open data corpora, respectively. For Josie, the index generation time is 336 and 50 hours, respectively.

7.1.1 Systems. To our best knowledge, there is no a multi-attribute join discovery system. We compare MATE and XASH, to multiple possible adaptations of single-column systems.

Single-Column Retrieval (SCR). We, adapted a single column index SCR for n-ary join discovery. It uses all the optimizations in Algorithm 1 including the table filtering strategies and initial column selection. Yet, SCR cannot utilize the super key: It fetches the candidate joinable rows and validates them through exact value comparisons in memory.

SCR Josie. Josie is the a state-of-the-art single-column join discovery algorithm introduced in literature. Josie leverages an index that maps values to columns, where as the columns are represented as sets. To infer the joinable rows we fall back on the SCR index.

Multi-Column Retrieval (MCR) [47]. In this approach, we fetch the PL items for each query column and intersect the results to discover the top-k joinable tables.

MCR Josie [47]. One can adapt Josie to discover multi-column joins by repeating it on every key column in Q, finding the most joinable tables per column, and then intersecting them and evaluating the tables that appear in all joinable results.

7.1.2 Hash Functions. We compare the filtering performance of MATE using different hash functions.

Bloom filter (BF). This is a variation of MATE that uses bloom filters (instead of XASH) with a fixed number of hash functions to generate the super key [5]. The BF implementation calculates the number of hash functions based on the average number of columns in the corpus tables. We use Murmur3 hash family as the base function in the BF implementation. In BF, the FP ratio can be calculate as: $FP = (1 - e^{-\frac{V \cdot H}{|a|}})^H$, where V is the number of values to be inserted and H is the number of hash functions [14]. To calculate the optimum number of hash functions, we set FP to 0. Doing so, the number of hashes can be calculated as: $H = \frac{|a|}{V} log2$. We set $V = 5$ for webtables and $V = 26$ for OD, using the average number of columns in the corresponding corpus.

Less Hashing Bloom Filter [19] (LHBF). This approach is an optimized version of the standard bloom filter that uses only two hash functions per value. We use the LessHash-BloomFilter python library, which is publicly available. [6]

Hash table (HT). It is the same as BF but uses one hash function.

Other hash functions. We also compare MATE to hash functions that are not followed by any post-processing of the hash values. We considered the following state-of-the-art hash functions: SimHash [5, 37], MD5 [35], Google's CityHash [34], and Murmur. We excluded hash functions based on SHA as they turned out to be similar but slower than the aforementioned hash functions.

[5]https://gist.github.com/malaikannan/22401e67b332f52651dcef8ea5039498
[6]https://github.com/garawalid/LH-BloomFilter

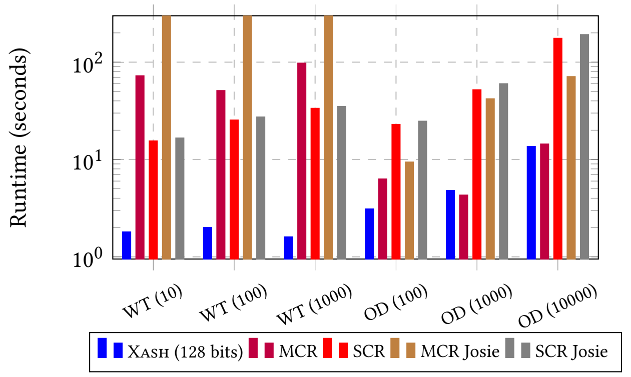


Figure 4: Runtime comparison between MATE and Baselines.

7.2 MATE VS. Baseline Systems

Figure 4 shows the runtime comparison MATE against SCR, MCR, and the corresponding JOSIE implementations in log scale. Without the row filtering optimization of MATE, the runtime for joinability calculation dominates the overall runtime. The depicted runtime results for baselines include the time required to discover the multi-column joins in-memory on top of the state-of-the-art single-attribute joinable table discovery systems. Note that there is an additional fetching cost that is negligible, i.e., in the order of milliseconds when we use the in-memory implementation like state-of-the-art [47]. However, the fetching cost can vary between 1 and 40 seconds when the data and the index has to be retrieved from disk. This is the case for the DWTC, which cannot fit into memory. The depicted runtime does not contain the fetching time as it is the same for both approaches. We observe that MATE outperforms baseline systems in almost all experiments: it is up to $61x$, $13x$, $9x$, $22x$ faster than MCR, SCR, MCR Josie, and SCR Josie respectively. This experiment also shows that no other baseline constantly performs better than the other approaches. For instance, SCR-based approaches are slower than their corresponding MCR-based systems for open data queries but on the large webtable corpus they underperform. Note that MCR Josie for webtables did not terminate in 7 days. The slower performance of the MCR approaches on webtable queries is because accessing the relevant PLs does not scale for the size of the webtable corpus. While MCR Josie performs similar to MCR in OD (100) its performance drops as the cardinality of the open data queries grows. The size of the input dataset correlates with the runtime. This is expected because join columns with higher cardinality initially match more PL items in the index which again need to be fetched and filtered.

In addition to the input table size, FPs play an important role in the number of comparisons and ultimately the runtime. For instance, in case of SCR, although OD (1k) and WT (1k) have similar query table sizes, the OD (1k) queries lead to 35% higher runtime than WT (1k). This increase in runtime is because of the large number of FPs for OD (1k) tables, i.e., $3M$ more FP rows compared to WT (1k).

Summary. (i) Mate is up to 100*x* faster than unary join discovery systems in discovering n-ary joins. (ii) Performance gain of Mate over SCR-based approaches depends on the number of FP rows.

7.3 Xash VS. Baseline Hash Functions

Table 2 depicts the runtime of Mate with different hash functions, including the proposed Xash. Note that all the competing hash functions benefit from all of Mate's optimizations and only differ in the applied hash function during row filtering.

In Table 2, we highlight the best performing approach of each row and hash size in **bold** font. We observe that Mate with Xash outperforms all the other baselines on all of the queries. Mate + Xash can be up to 10*x* faster than Mate + BF, which is the second most efficient baseline on average. The cells showing this speedup are highlighted with red. BF's under-performance is rooted in its higher collision rate compared to Xash. Besides, by checking the length attribute that we use in Xash to generate the hash results, we can drop many of the non-joinable rows without evaluating all the bits in the super keys. This feature gives Mate a superiority over BF in runtime even in the cases with similar or lower FP rates. Another interesting insight is that Mate + BF is faster than Mate + HT in most of the cases and this is because BF is able to leverage more bits in the hash space to encode the join keys than a hash table that leverages only a single bit to hash each key value.Table 2 shows that Xash performs also better than LHBF in all of the cases. LHBF leverages fewer hash functions in comparison to BF with statistically similar FP probability. However, similar to HT, fewer hashing does not necessarily lead to a better performance in discovering the joinable tables. In our experiments LHBF only performs better on larger hash sizes for webtable queries, but in the remaining cases, LHBF is less efficient than BF.

The results also show that although super keys based on the standard hash functions, MD5, CityHash, SimHash, and Murmur, lead to overall performance gains over the naive SCR approach, all of them clearly fall behind Xash. This is because they are not optimized to identify subset relationships that we encode via bitwise OR masking. They generate too many 1 bits (on average 50%) in the hash results, which leads to a higher number of FPs. Thus, if a table contains six columns the aggregation of six hash results will on average turn 98% of the super key to 1s, which would make the super key highly ineffective in masking.

Further, the experiments show that in most of the cases, a larger hash size (512 bits) leads to a faster join discovery than the smaller versions. However, in cases such as WT (100) and Simhash, the larger hash size results in slower discovery. The experiments that lead to slower runtime compared to their smaller hash size are marked as blue cells. When the FP rate is similar for two hash sizes, the approach with the larger hash size will result in similar or higher runtime. This is because the bit-wise OR operation is more expensive for larger hash values. For instance, for WT (100) query tables and SimHash, the difference between the FP rate is almost zero, therefore, the runtime overhead of 512-bit hash causes worse performance. In the more common case that a larger hash size reduces the number of FPs the resulting runtime is also lower. For example, for the OD (100) datasets, the larger hash size for Xash results in 85% fewer FPs on average, leading to 48% speed-up.

Finally, we also observed that Mate can indeed find interesting joinable tables based on composite keys. Searching for the top joinable tables to Kaggle Movie dataset based on a single-column join and "Movie Title" key columns, none of the top-10 tables contains more than one additional column, containing float numbers. However, using our multi-column join discovery approach and query columns <"Director name", "Movie Title">, we obtain a table with 8 columns worth of information, including the plot of the movie, actor names, etc. For the *Kaggle Airline* dataset with join keys "Airline name" and "Country" Mate obtains a table representing the airports in which the airlines operate flights in the given countries.
Summary. (i) The syntactic feature extraction in Xash leads to a faster multi-attribute join discovery compared to the baselines; (ii) the standard hash functions are not ideal for n-ary joins due to their uniform distribution property that results in too many 1 bits; and (iii) larger hash sizes result in higher calculation overhead.

7.4 FP Rate

There is a direct relationship between the table discovery runtime and the FP rate of the hash function used. We measure the precision as the ratio of true positives (TP) over both TPs and FPs: $precision = \frac{TP}{TP+FP}$. Precision normalizes the TP-rate across datasets.

Table 3 shows the results of this experiment. Due to space reasons in this experiment, we only show the results for 128- and 512-bit hash sizes. Note that we observe the same trends as the other hash sizes. Like runtime, precision increases with larger hash sizes in most of the cases. This is due to the fact that hash functions can scatter the values in a larger range and this leads to lower collision rate between super keys. On average, Mate + Xash achieves the highest precision compared to all the other approaches both in 128- and 512-bit hash spaces. Xash achieves up to 25% higher precision than BF. However, in five cases, BF outperforms Xash. In these experiments the difference between BF and Xash is only 4%. Mate's runtime superiority regardless of its precision is because of the design of Xash: the length feature of the cell value is located on the left-most segment of the array. Thus, the bit-wise OR operation can skip a table row before even getting to the character features of the cell values ensuring that this happens in the very first bit-segment of the hash. If the length of the key value has never appeared in the candidate row, Mate moves to the next row. In baseline approaches, such as BF, the 0-bits are randomly scattered. Therefore, step-wise filtering might take longer to find the contradicting bits. Xash displays the smallest standard deviation showing its robustness in achieving high precision. In general, the precision achieved by LHBF is consistent with its runtime performance compared to the BF approach. LHBF achieves higher precision and ultimately, lower runtime for webtable queries with 512-bit hash sizes but in the remaining experiments BF outperforms LHBF.
Summary. (i) The precision of an approach correlates with its runtime. (iii) The segmentation of bits in Xash leads to faster join discovery even in cases where precision rates are similar.

7.5 Mate In Depth

7.5.1 Top-k. The number of top-*k* joinable tables changes the stopping criteria of the system (see table filtering step in Section 6). Larger *k* requires more tables to be evaluated. In this experiment,

Table 2: Runtime experiment (seconds). blue cells represent the experiments where the larger hash performs worse. Red cells show the maximum performance gain of Mate over BF.

Dataset	*SCR*	*MD5*	*Murmur*	*City*	*SimHash*			*HT*			*BF*			*LHBF*			*Xash*		
		128	**128**	**128**	**128**	**256**	**512**	**128**	**256**	**512**	**128**	**256**	**512**	**128**	**256**	**512**	**128**	**256**	**512**
WT (10)	20.6	12.5	12.6	12.6	11.1	10.1	12.5	6.4	7.3	7.6	4.2	4.2	10.3	4.1	3.6	2.2	**1.8**	**1.5**	**1.4**
WT (100)	24.5	15.3	14.8	15.3	13.4	8.5	17.2	9.5	10.1	12.4	4.7	5.0	16.8	5.6	3.6	4.2	**1.6**	**1.8**	**2.0**
WT (1k)	32.7	20.2	19.8	19.8	16.3	10.0	17.0	13.4	13.8	13.4	5.6	5.5	16.7	7.6	4.1	4.6	**1.6**	**1.5**	**1.6**
OD (100)	247.6	212.1	224.2	194.5	102.9	69.1	56.5	24.0	16.4	10.8	11.0	4.8	3.4	30.1	9.5	8.3	**6.5**	**3.5**	**3.1**
OD (1k)	165.8	146.8	152.3	138.6	90.0	73.6	62.6	25.4	21.6	16.5	14.4	7.6	5.6	39.2	11.3	13.0	**8.7**	**4.5**	**4.8**
OD (10k)	256.7	203.9	202.1	190.6	145.6	127.0	103.5	49.6	44.0	31.9	27.7	17.8	14.3	79.2	27.0	21.8	**17.6**	**13.4**	**13.6**
Kaggle	297.0	97.6	104.2	104.4	76.6	57.3	41.0	34.8	30.3	25.9	19.7	14.8	13.1	37.8	27.1	21.8	**15.2**	**12.0**	**12.3**
School	873.6	772.7	772.9	742.8	593.7	444.0	307.2	54.3	38.3	31.0	24.2	19.0	19.0	33.8	25.3	22.7	**20.0**	**17.9**	**17.6**

Table 3: Precision experiment.

Dataset	*MD5*	*CityHash*	*SimHash*		*HT*		*BF*		*LHBF*		*Xash*	
	128	**128**	**128**	**512**	**128**	**512**	**128**	**512**	**128**	**512**	**128**	**512**
WT (10)	0.27±0.39	0.25±0.38	0.28±0.40	0.31±0.42	0.34±0.43	0.28±0.43	0.44±0.46	0.40±0.43	0.44±0.45	0.61±0.46	**0.57±0.46**	**0.88±0.26**
WT (100)	0.27±0.40	0.27±0.40	0.27±0.40	0.27±0.39	0.34±0.41	0.38±0.41	0.46±0.44	0.34±0.41	0.45±0.43	0.63±0.44	**0.61±0.43**	**0.93±0.22**
WT (1k)	0.24±0.36	0.24±0.36	0.25±0.37	0.28±0.35	0.40±0.37	0.42±0.36	0.59±0.39	0.32±0.35	0.52±0.38	0.78±0.33	**0.77±0.34**	**0.98±0.10**
OD (100)	0.27±0.38	0.28±0.39	0.28±0.38	0.32±0.41	0.43±0.40	0.55±0.41	**0.56±0.41**	0.79±0.34	0.45±0.43	0.67±0.35	0.52±0.41	**0.80±0.34**
OD (1k)	0.32±0.40	0.32±0.40	0.32±0.39	0.41±0.41	0.47±0.40	0.59±0.41	**0.61±0.40**	0.85±0.28	0.44±0.34	0.63±0.39	0.53±0.41	**0.86±0.28**
OD (10k)	0.27±0.38	0.28±0.38	0.28±0.37	0.34±0.39	0.42±0.39	0.56±0.39	**0.59±0.40**	**0.87±0.27**	0.40±0.42	0.66±0.40	0.52±0.42	0.82±0.32
School	0.00±0.00	0.00±0.00	0.00±0.00	0.00±0.00	0.01±0.01	0.03±0.02	0.07±0.03	**1.00±0.00**	0.02±0.01	0.24±0.08	**0.43±0.11**	0.96±0.02
Kaggle	0.09±0.18	0.09±0.17	0.12±0.21	0.20±0.31	0.25±0.31	0.44±0.37	0.40±0.41	0.64±0.38	0.33±0.34	0.63±0.36	**0.64±0.34**	**0.93±0.10**
Average	0.22±0.31	0.22±0.31	0.23±0.32	0.27±0.34	0.33±0.34	0.41±0.35	0.47±0.37	0.65±0.31	0.38±0.35	0.61±0.35	**0.57±0.37**	**0.90±0.21**

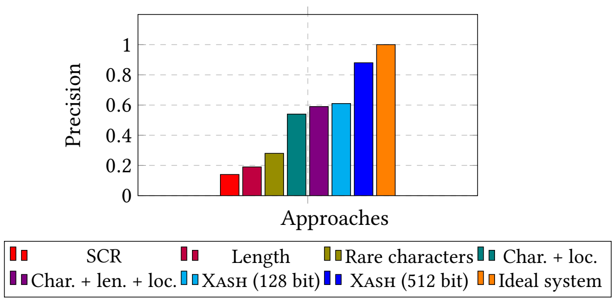


Figure 5: The influence of Xash components on Precision.

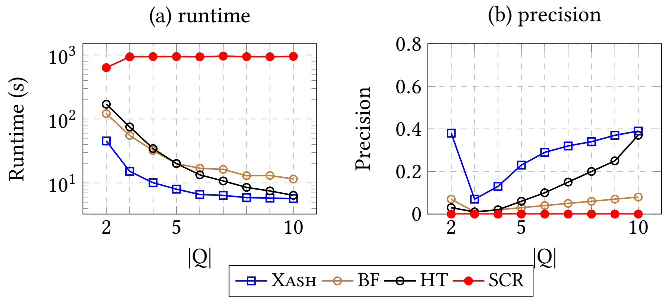


Figure 6: Key size experiment.

we measure the precision of Mate with different hash functions and vary k from 2 to 20. We ran the experiment with the WT (100) datasets as the input dataset against the DWTC corpus. Other variables, such as key columns, are fixed in this experiment. In this expeirment, Mate + Xash achieves the highest precision in comparison to other approaches for all k values. Increasing k, the precision for Xash increases 4%, while the precision remains the same for BF. The other hash functions lead to a slight precision cut encountering new tables with lower candidate joinable rows. This shows that Xash has higher ability to filter non-related rows compared to the given baselines. According to our observations on the candidate tables, this variation in precision occurs when the candidate tables contain more columns than average.

7.5.2 *Xash components.*

We now evaluate the impact of each feature used in Xash on the average precision and the FP rate of Mate. We use the WT (100) datasets in this experiment with the same setup as in the previous experiments. According to the results in Figure 5, encoding the *character and its location* has higher filtering power than the *length* feature. The difference between Xash and *character + length + location* is the rotation operation. In particular, we observe that the rotation filters 20% of the remaining FPs undiscovered in *character + length + location*. This shows that the rotation plays an important role in pruning FPs.

7.5.3 *Join-Key Size.*

Here, we evaluate the scalability of Mate in the existence of different join-key sizes, i.e., the number of columns in the composite key of the input table. Due to the limited number of tables with high dimensional composite key, we ran this experiment with a random dataset from the German Open Data corpus with up to 10 columns that can form a composite key (out of 33 columns). Figure 6 (a) and 6 (b) depict the runtime and precision results for different key sizes. Increasing the number of columns, we observe that the runtime for Mate constantly reduces because the FP rate constantly decreases. However, this does not imply a constant increase in precision because increasing the number of key columns changes the ratio of joinable and non-joinable rows. In our experiment, moving from a composite key with two columns to a three-column key, 97% of the joinable rows become non-joinable due to the newly introduced key column. Thus, the filter is confronted with a larger number of FPs, some of which come through. From key size 4 upwards, precision increases again as expected. The runtime gain for larger key sizes is because of two reasons: *(1)* With more columns in the query join key, there will be more 1-bits in the query super key, which makes it harder to mask. *(2)* Increasing the size of the composite key leads to fewer joinable table rows. Therefore, it is more likely that the second table filtering rule drops the candidate tables without evaluating the remaining rows.

7.5.4 Initial column selection. We evaluate our heuristic to select the initial query column by comparing the number of retrieved PL items in Mate to four other baselines: *(i)* The column order. In this approach, the initial column will be the first query column according to the column order inside the table. *(ii)* The longest string (TLS). In this approach, the system selects the column that contains the longest cell value as the initial column. *(iii)* The worst-case scenario. A hypothetical approach that always picks the worst column that returns the larger number of PL items. *(iv)* Best, i.e. ground truth, which chooses the column that filters most. The best- and the worst-case scenarios provide lower and upper bounds for our experiment. This experiment is done using the OD (10k) tables. In this experiment our cardinality-based heuristic outperfmed the other heuristics. It retrieved on average only 179 PLs compared to Column Order, TLS, and the worst-case scenario with 202, 248, and 728 Pls, respectively. The optimal number of PLs based on ground truth was 83. The heuristic used in Mate performs better because of the fact that the number of PL items per cell value follows the power-law distribution. There is a small set of values that have a large number of PL items but most of the values lead to a similar number of PL items (The average number is 12).

8 RELATED WORK

N-ary joinable table search is not a well-researched area and a very limited number of works have focused on multi-attribute joins discovery. Li et al.'s index design [24] is one of these works. They introduce a prefix tree index to calculate the joinability score between two given tables for n-ary joins. The drawback of this approach is that it is not scalable to data lakes. The prefix tree-based approach assumes that the one-to-one mapping between the composite key columns and the columns in the candidate tables is apriori known. Mate, on the other hand, leverages an index structure that does not require the user to provide the mappings.

There is a large number of research papers that address joinable tables discovery problem based on unary keys [8, 11, 15, 41, 42, 45–47, 49]. Mate extends the standard single-attribute inverted index used in these state-of-the-art systems and applies a fast single-operation filtering approach to be able to detect joinable table rows without actual value comparison. Any single attribute inverted index can be extended with the super key used in Mate.

For spatial indices, the goal of an ideal n-ary join discovery is to map multiple dimensions into an easy to search index. However, spatial index structures are a very special case compared to the problem we tackle in this paper. The spatial indices e.g. KD tree [2], KDB tree [36], R+ tree [39], Geohash [25], grid files [31], suffer from three drawbacks: 1) the indexed data points always have fixed dimensions, such as latitude and longitude. 2) These dimensions have a fixed order. For instance, latitude is always compared to the latitude of the other points. In other words, there is a pre-defined mapping between the dimensions. However, in join discovery, key values can appear in arbitrary columns and comparing the right values is not a straightforward task. 3) These indices are designed to work with numerical values. Mate on the other hand does not have any of the aforementioned assumptions. It discovers the joinable table rows with an arbitrary number of columns. Also, Mate leverages Xash and discovers the key values regardless of their position in the candidate rows. Besides, Xash is applicable for numerical and alphabetical column values.

Data enrichment solutions attempt to join additional tables to improve the downstream machine learning accuracy. Data enrichment solutions also either require pre-defined mappings between joinable columns [7, 21], which needs the user input on every single external table, or they focus on single column joins [8, 11, 13, 41, 47]. Mate calculates the joinability of the candidate tables without any apriori knowledge, e.g. mapping information. This allows Mate to scale to any number of external tables.

Inclusion dependency (IND) discovery [9, 20, 33] is a research direction where the goal is to discover foreign keys in the database [33]. The focus is on identifying fitting column combinations across all datasets. In data discovery, we are interested in the top-k tables with the most fitting INDs.

9 CONCLUSION AND FUTURE WORK

In this paper, we tackled the problem of n-ary or multi-attribute join search. We proposed a hash-based filtering approach Mate that removes redundant table rows from further joinability calculation processes to increase the scalability of the join discovery. We proposed Xash, a hash function that leverages syntactic features of the cell values that ultimately lead to the lowest FP rates. We showed that Mate is more effective and efficient than state-of-the-art systems. We focused on the equi-joins. Because Xash uses syntactic features including the character and length features of the cell values, it has the potential to discover similarity joins as well. According to our observations, the false positives caused by Xash were those that are syntactically similar to the actual key values, e.g., discovering the composite key <"brooklyn", "cambridge"> instead of <"brooklyn", "bay ridge">. Another future direction is to reason about hashing short key values. Xash cannot use its optimal potential if cell values are too short.

ACKNOWLEDGMENTS

This project has been supported by the German Research Foundation (DFG) under grant agreement 387872445 and the German Ministry for Education and Research as BIFOLD — "Berlin Institute for the Foundations of Learning and Data" (01IS18025A and 01IS18037A).

REFERENCES

[1] Ziawasch Abedjan, John Morcos, Michael N Gubanov, Ihab F Ilyas, Michael Stonebraker, Paolo Papotti, and Mourad Ouzzani. 2015. Dataxformer: Leveraging the Web for Semantic Transformations.. In *CIDR*.

[2] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

[3] Alex Bogatu, Alvaro A. A. Fernandes, Norman W. Paton, and Nikolaos Konstantinou. 2020. Dataset Discovery in Data Lakes. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020.* IEEE, 709–720. https://doi.org/10.1109/ICDE48307.2020.00067

[4] Michael J. Cafarella, Alon Y. Halevy, and Nodira Khoussainova. 2009. Data Integration for the Relational Web. *Proc. VLDB Endow.* 2, 1 (2009), 1090–1101. https://doi.org/10.14778/1687627.1687750

[5] Moses S Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing.* 380–388.

[6] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. 2006. A primitive operator for similarity joins in data cleaning. In *22nd International Conference on Data Engineering (ICDE'06).* IEEE, 5–5.

[7] Nadiia Chepurko, Ryan Marcus, Emanuel Zgraggen, Raul Castro Fernandez, Tim Kraska, and David Karger. 2020. ARDA: Automatic Relational Data Augmentation for Machine Learning. *PVLDB* 13, 9 (2020).

[8] Anish Das Sarma, Lujun Fang, Nitin Gupta, Alon Halevy, Hongrae Lee, Fei Wu, Reynold Xin, and Cong Yu. 2012. Finding related tables. In *SIGMOD.* 817–828.

[9] Fabien De Marchi, Stéphane Lopes, and Jean-Marc Petit. 2009. Unary and n-ary inclusion dependency discovery in relational databases. *Journal of Intelligent Information Systems* 32, 1 (2009), 53–73.

[10] Yuyang Dong, Kunihiro Takeoka, Chuan Xiao, and Masafumi Oyamada. 2021. Efficient joinable table discovery in data lakes: A high-dimensional similarity-based approach. In *2021 IEEE 37th International Conference on Data Engineering (ICDE).* IEEE, 456–467.

[11] Julian Eberius, Maik Thiele, Katrin Braunschweig, and Wolfgang Lehner. 2015. Top-k entity augmentation using consistent set covering. In *SSDBM.* 8.

[12] Mahdi Esmailoghli and Ziawasch Abedjan. 2020. CAFE: Constraint-Aware Feature Extraction from Large Databases. (2020).

[13] Mahdi Esmailoghli, Jorge-Arnulfo Quiané-Ruiz, and Ziawasch Abedjan. 2021. COCOA: COrrelation COefficient-Aware Data Augmentation. EDBT.

[14] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. 2000. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM transactions on networking* 8, 3 (2000), 281–293.

[15] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, and Michael Stonebraker. 2018. Aurum: A data discovery system. In *ICDE.* IEEE, 1001–1012.

[16] Raul Castro Fernandez, Jisoo Min, Demitri Nava, and Samuel Madden. 2019. Lazo: A cardinality-based method for coupled estimation of jaccard similarity and containment. In *2019 IEEE 35th International Conference on Data Engineering (ICDE).* IEEE, 1190–1201.

[17] Arvid Heise, Jorge-Arnulfo Quiané-Ruiz, Ziawasch Abedjan, Anja Jentzsch, and Felix Naumann. 2013. Scalable discovery of unique column combinations. *Proceedings of the VLDB Endowment* 7, 4 (2013), 301–312.

[18] Lan Jiang and Felix Naumann. 2019. Holistic primary key and foreign key detection. *Journal of Intelligent Information Systems* (2019), 1–23.

[19] Adam Kirsch and Michael Mitzenmacher. 2006. Less Hashing, Same Performance: Building a Better Bloom Filter. In *Algorithms - ESA 2006, 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006, Proceedings (Lecture Notes in Computer Science)*, Yossi Azar and Thomas Erlebach (Eds.), Vol. 4168. Springer, 456–467. https://doi.org/10.1007/11841036_42

[20] Andreas Koeller and Elke A Rundensteiner. 2003. Discovery of high-dimensional inclusion dependencies. In *Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405).* IEEE, 683–685.

[21] Arun Kumar, Jeffrey Naughton, Jignesh M. Patel, and Xiaojin Zhu. 2016. To join or not to join?: Thinking twice about joins before feature selection. In *SIGMOD.* 19–34.

[22] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandier, Lyric Doshi, and Chuck Bear. 2012. The vertica analytic database: C-store 7 years later. *PVLDB* 5, 12 (2012), 1790–1801.

[23] Oliver Lehmberg and Christian Bizer. 2017. Stitching web tables for improving matching quality. *PVLDB* 10, 11 (2017), 1502–1513.

[24] Guoliang Li, Jian He, Dong Deng, and Jian Li. 2015. Efficient similarity join and search on multi-attribute data. In *SIGMOD.* 1137–1151.

[25] Jiajun Liu, Haoran Li, Yong Gao, Hao Yu, and Dan Jiang. 2014. A geohash-based index for spatial data management in distributed memory. In *2014 22nd International Conference on Geoinformatics.* 1–4. https://doi.org/10.1109/GEOINFORMATICS.2014.6950819

[26] Mark S Mayzner and Margaret Elizabeth Tresselt. 1965. Tables of single-letter and digram frequency counts for various word-length and letter-position combinations. *Psychonomic monograph supplements* (1965).

[27] Holger J Meyer, Hannes Grunert, Tim Waizenegger, Lucas Woltmann, Claudio Hartmann, Wolfgang Lehner, Mahdi Esmailoghli, Sergey Redyuk, Ricardo Martinez, Ziawasch Abedjan, et al. [n.d.]. Particulate Matter Matters-The Data Science Challenge@ BTW 2019. *Datenbank-Spektrum* ([n. d.]), 1–18.

[28] Renée J Miller. 2018. Open data integration. *Proceedings of the VLDB Endowment* 11, 12 (2018), 2130–2139.

[29] Fatemeh Nargesian, Ken Q Pu, Erkang Zhu, Bahar Ghadiri Bashardoost, and Renée J Miller. 2020. Organizing data lakes for navigation. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data.* 1939–1950.

[30] Fatemeh Nargesian, Erkang Zhu, Ken Q Pu, and Renée J Miller. 2018. Table union search on open data. *PVLDB* 11, 7 (2018), 813–825.

[31] Jürg Nievergelt, Hans Hinterberger, and Kenneth C Sevcik. 1984. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems (TODS)* 9, 1 (1984), 38–71.

[32] Aslihan Özmen, Mahdi Esmailoghli, and Ziawasch Abedjan. 2021. Combining Programming-by-Example with Transformation Discovery from large Databases. 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 (LNI)*, Kai-Uwe Sattler, Melanie Herschel, and Wolfgang Lehner (Eds.), Vol. P-311. Gesellschaft für Informatik, Bonn, 313–324. https://doi.org/10.18420/btw2021-16

[33] Thorsten Papenbrock, Sebastian Kruse, Jorge-Arnulfo Quiané-Ruiz, and Felix Naumann. 2015. Divide & conquer-based inclusion dependency discovery. *Proceedings of the VLDB Endowment* 8, 7 (2015), 774–785.

[34] G Pike and J Alakuijala. 2017. The CityHash family of hash functions (2011). https://github.com/google/cityhash.

[35] Ronald Rivest and S Dusse. 1992. The MD5 message-digest algorithm.

[36] John T Robinson. 1981. The KDB-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data.* 10–18.

[37] Caitlin Sadowski and Greg Levin. 2007. Simhash: Hash-based similarity detection. *Technical report, Google* (2007).

[38] Aécio Santos, Aline Bessa, Fernando Chirigati, Christopher Musco, and Juliana Freire. 2021. Correlation sketches for approximate join-correlation queries. In *Proceedings of the 2021 International Conference on Management of Data.* 1531–1544.

[39] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. 1987. *The R+-Tree: A Dynamic Index for Multi-Dimensional Objects.* Technical Report.

[40] Jonas Traub, Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. [n.d.]. Agora: Bringing Together Datasets, Algorithms, Models and More in a Unified Ecosystem [Vision]. ([n. d.]).

[41] Chuan Xiao, Wei Wang, Xuemin Lin, and Haichuan Shang. 2009. Top-k set similarity joins. In *ICDE.* IEEE, 916–927.

[42] Mohamed Yakout, Kris Ganjam, Kaushik Chakrabarti, and Surajit Chaudhuri. 2012. Infogather: entity augmentation and attribute discovery by holistic matching with web tables. In *SIGMOD.* 97–108.

[43] Minghe Yu, Guoliang Li, Dong Deng, and Jianhua Feng. 2016. String similarity search and join: a survey. *Frontiers of Computer Science* 10, 3 (2016), 399–417.

[44] Dan Zhang, Yoshihiko Suhara, Jinfeng Li, Madelon Hulsebos, Çagatay Demiralp, and Wang-Chiew Tan. 2020. Sato: Contextual Semantic Type Detection in Tables. *Proc. VLDB Endow.* 13, 11 (2020), 1835–1848. http://www.vldb.org/pvldb/vol13/p1835-zhang.pdf

[45] Meihui Zhang and Kaushik Chakrabarti. 2013. InfoGather+ semantic matching and annotation of numeric and time-varying attributes in web tables. In *SIGMOD.* 145–156.

[46] Yi Zhang and Zachary G. Ives. 2020. Finding Related Tables in Data Lakes for Interactive Data Science. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020,* David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1951–1966. https://doi.org/10.1145/3318464.3389726

[47] Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée J Miller. 2019. JOSIE: Overlap Set Similarity Search for Finding Joinable Tables in Data Lakes. In *SIGMOD.* 847–864.

[48] Erkang Zhu, Yeye He, and Surajit Chaudhuri. 2017. Auto-Join: Joining Tables by Leveraging Transformations. *Proc. VLDB Endow.* 10, 10 (2017), 1034–1045. https://doi.org/10.14778/3115404.3115409

[49] Erkang Zhu, Fatemeh Nargesian, Ken Q. Pu, and Renée J. Miller. 2016. LSH Ensemble: Internet-Scale Domain Search. *Proc. VLDB Endow.* 9, 12 (2016), 1185–1196. https://doi.org/10.14778/2994509.2994534