MatchCatcher: A Debugger for Blocking in Entity Matching

Han Li[1], **Pradap Konda**[1], **Paul Suganthan G.C.**[1], **AnHai Doan**[1],
Benjamin Snyder[2], **Youngchoon Park**[3], **Ganesh Krishnan**[4], **Rohit Deep**[4], **Vijay Raghavendra**[4]
[1]University of Wisconsin-Madison, [2]Amazon, [3]Johnson Controls, [4]@WalmartLabs

ABSTRACT

Blocking is a fundamental step in entity matching (EM). Much work has examined the design and runtime of blockers. However, very little if any work has examined the problem of debugging blocking accuracy. In practice, blockers' accuracy can vary drastically, and using an accurate blocker is critical for many EM applications. To address this problem, we describe the MatchCatcher solution. Given two tables to be matched and a blocker, MatchCatcher finds matches killed off by the blocker, so that the user can examine these matches to understand how well the blocker does accuracy-wise and what can be done to improve its accuracy. We show how to quickly find such matches using string similarity joins, iterative user engagement, rank aggregation, and active/online learning. Extensive experiments show that MatchCatcher is highly effective in helping users develop blockers, can help improve accuracy of even the best blockers manually created or automatically learned. MatchCatcher has been open sourced and used by 300+ students in data science class projects and 7 teams at 6 organizations.

1 INTRODUCTION

Entity matching (EM) finds data instances referring to the same real-world entity [6, 14], such as tuples (Dave Smith, San Francisco, CA) and (David Smith, S.F., CA). This problem is critical for many Big Data and data science applications.

When doing EM, we often must perform blocking. Consider for example matching two tables A and B. Real-world tables often have hundreds of thousands, or millions, of tuples. Trying to match all tuple pairs in $A \times B$ is practically infeasible. So we often perform a step called *blocking* which uses domain heuristics to quickly drop many pairs judged obviously non-matched (e.g., person tuples that do not have the same state). The next step, called *matching*, matches the remaining pairs, using rule- or learning-based techniques. Blocking can greatly reduce the number of pairs considered in the matching step, drastically reducing the total EM time. As a result, virtually all real-world EM applications use blocking.

Numerous blocking methods have been developed [6]. For example, *hash blocking* drops all tuple pairs that do not have the same hash value, using a predefined hash function. This method is popular because it is easy to understand and fast. Other methods include sorted neighborhood, overlap, phonetic, rule-based, etc. (see Section 2).

Given two tables A and B to match, we often want a blocker Q that is *fast*, *selective*, and *accurate*. "Fastness" is measured by the time to apply Q to A and B to produce a set of tuple pairs C. "Selectivity" is typically measured as the ratio $|C|/|A \times B|$. "Accuracy" is typically measured as the fraction of *true matches* surviving blocker Q, i.e., $|M \cap C|/|M|$, where M is the set of (unknown) true matches in $A \times B$. As such, it is also referred to as *recall*.

In practice, blockers can vary drastically in recall, and using a blocker with high recall is critical for many EM applications (see Section 2). Yet today there is still no good way to develop such blockers. For example, given the popularity of hash blockers, suppose we have decided to use a hash blocker Q on two tables. While fast, Q may have low recall if the attribute values to be hashed are dirty, misspelt, missing, or have many natural variations (e.g., "New York", "NY", "NYC"). A common way to address this problem is to use multiple hash blockers and take the union of their outputs, to maximize recall. However, even in this case, the recall can still be quite low. For instance, a recent work [8] describes two real-world datasets where extensive effort at combining hash blockers achieves only 38.8% and 72.6% recall. Such low recalls are simply unacceptable for many EM applications. To improve recall, we can revise the current hash blockers, replace some of them, or adding more blockers (of the non-hash types). *To do any of these, however, we need a way to understand whether the current blocker has low recall, and if so, then what the possible problems are, so that we can improve it.*

The MatchCatcher Solution: In this paper we take the first step toward solving the above problems. We describe MatchCatcher, a solution to debug blocker accuracy. Given two tables A and B to be matched and a blocker Q, MatchCatcher attempts to find matches that are "killed off" by Q, i.e., those that do not survive the blocking step. We can examine these matches to see if they are indeed true matches, and if so, then why they get killed off by Q. This tells us whether Q has low recall, and if so, then how to improve it. The following example illustrates our solution:

Example 1.1. Consider matching tables A and B in Figure 1.a. Suppose a user U begins by creating a blocker Q_1 that keeps only tuple pairs sharing the same value for "City". Figure 1.b shows this blocker as Q_1: $a.City = b.City$. (This is attribute-equivalence blocking, a special type of hash blocking.) Applying Q_1 to A and B produces a set of tuple pairs C_1 (see Figure 1.b).

User U wants to know if blocker Q_1 kills off too many true matches. To answer this, U applies MatchCatcher, *which operates in iterations. In the first iteration,* MatchCatcher *shows the user n tuple pairs judged most likely to be matches killed off by Q_1. These pairs are listed on Figure 1.b, under "Debugger Output, Iter 1" (here $n = 3$).*

User U finds that the first two pairs, (a_1, b_1) and (a_3, b_2), are indeed true matches (shown in red color on the figure). A closer examination reveals that they do not survive blocking because their "City" values do not match due to misspellings and abbreviation, e.g., "Altanta" vs. "Atlanta", "New York" vs. "NY".

Next, U wants to know if there are any more true matches. Toward this goal, U flags the true matches in the first iteration (i.e., the above two pairs). MatchCatcher *uses this feedback to find the next n pairs judged most likely to be killed-off matches, then shows those pairs in the second iteration (see Figure 1.b, under "Iter 2").*

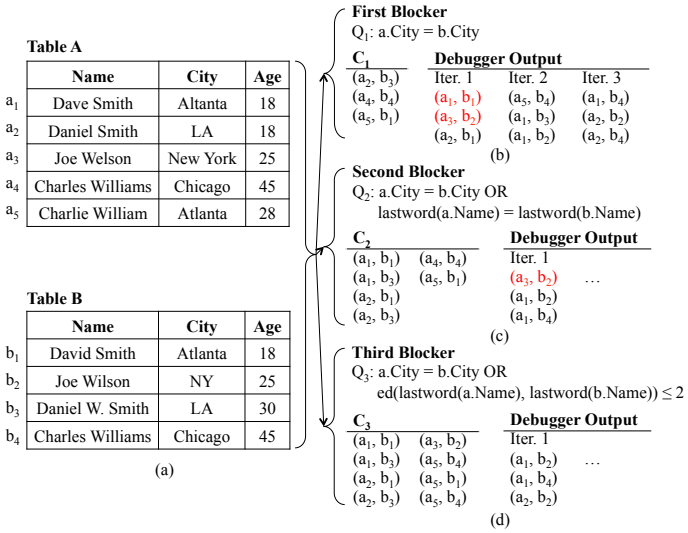

Figure 1: An example to illustrate MatchCatcher

U finds no true matches in this iteration, as well as in the third iteration.

At this point, U decides to stop looking for more killed-off matches, to focus on revising blocker Q_1 to improve its recall. U observes that the problem with pair (a_1, b_1), which disagree on "City", can be fixed by adding a new hash blocker that blocks on the last word of "Name", i.e., keeps a tuple pair if they agree on this word (which is typically the last name). Figure 1.c shows Q_2, the revised blocker, which is the union of two hash blockers.

Invoking MatchCatcher *for Q_2 produces the list shown under "Debugger Output, Iter 1" in Figure 1.c. This list shows that while the new blocker Q_2 successfully keeps (a_1, b_1), it still kills off (a_3, b_2), a true match. A closer examination reveals that this is due to a mispelt last word: "Welson" vs. "Wilson".*

To fix such misspelling problems, U decides to keep a tuple pair if the last words of "Name" are very similar, e.g., within an edit distance of 2. This produces blocker Q_3 in Figure 1.d. Here, the hash blocker $lastword(a.Name) = lastword(b.Name)$ has been replaced by $ed(lastword(a.Name), lastword(b.Name) \leq 2$, a more general blocker where "ed" computes the edit distance. Invoking MatchCatcher *for Q_3 brings back no true matches, even after several iterations. Thus, user U stops, deciding to use Q_3 as the final blocker for A and B.*

It is important to emphasize that MatchCatcher works with any of the current blocker types. Indeed, it requires as input only the two tables A and B and the set C resulting from applying the target blocker to the tables. MatchCatcher thus is *blocker independent*. We intentionally designed MatchCatcher this way to maximize its real-world applicability, i.e, to make sure that no matter which blocker a user has created, he or she can use MatchCatcher. Subsequent work will examine extending MatchCatcher to exploit the particularities of a specific blocker type.

Further, MatchCatcher does not estimate the *actual* recall, i.e., the fraction of matches surviving blocking. Doing so would require it to know the set of true matches in $A \times B$, which would be solving the EM problem itself! Indeed, MatchCatcher does *not* attempt to match A and B. Instead, *its goal is to quickly find a large set of plausible matches killed off by the blocker and bring them to the user's attention*, so that the user can examine them to find true matches, get a sense about whether the blocker kills off too many such matches, and if so, what the problems are, so that he/she can fix them. Section 6 shows that real-world users indeed find MatchCatcher very helpful in answering these questions.

Challenges: While promising, developing MatchCatcher raises difficult challenges. First, we must quickly search the vast space $D = A \times B - C$ (where C is the blocker's output) to find plausible matches killed off by the blocker, and we must do so without materializing D. This search is further complicated by the fact that at this point MatchCatcher does not even know what it means to be a match (only the user knows). To address these problems, we observe that matching tuples tend to have similar values for certain attributes (e.g., Name, City). So we convert each tuple into a string that concatenates these attributes, e.g., converting tuple a_1 of Table A in Figure 1.a into "Dave Smith Altanta". We then perform a *top-k string similarity join (SSJ)* to find the k tuple pairs with the highest score with respect to these strings, and output these pairs as plausible matches. The state-of-the-art solution for top-k SSJs [34] proves too slow for our interactive setting. So we develop a new solution that is significantly faster.

Second, to find as many plausible matches as possible, we need to repeat the above procedure, but for different sets of attributes (e.g., find tuple pairs that are similar with respect to Name only, City only, both Name and City, etc.). We cannot consider all such sets, called *configs*, as there are too many. So we develop a solution to find a good set of configs.

Third, we must perform multiple related top-k SSJs, one for each config. This raises the challenge of how to perform them *jointly* across the configs. We develop an efficient solution that perform them in parallel on multiple cores yet reuse computations across the joins.

Finally, top-k SSJs over the configs produce a large set E of plausible matches (e.g., in the thousands). We cannot realistically expect the user to examine all of these matches. So we develop a solution that uses rank aggregation and active/online learning to rank the pairs in E, show the top n pairs to the user, ask him/her to identify the true matches, use this feedback to rerank the pairs, and so on, until the user has been satisfied or a stopping condition is reached. In summary, we make the following contributions:

- We show that debugging blocker accuracy is critical for EM.
- We describe MatchCatcher. As far as we know, this is the first in-depth solution to address the above problem. Our solution advances the state of the art in top-k string similarity joins, and exploits active/online learning to effectively engage with the user.
- Over the past two years, MatchCatcher has been successfully used by 300+ students in data science projects and by 7 teams at 6 organizations. We briefly report on this experience. We also describe extensive experiments showing that MatchCatcher is highly effective in helping users develop blockers, and that it can help improve the accuracy of even the best blockers manually created or automatically learned.

2 DEBUGGING BLOCKER ACCURACY

In this section we show that debugging blocker accuracy is critical for EM, discuss the limitations of current solutions, then provide an overview of the MatchCatcher solution.

Entity Matching (EM): This problem has received significant attention (see [6, 14, 30] for recent books and surveys). Many EM scenarios exist, e.g., matching two tables, matching within a table,

matching a table with a knowledge base, etc. [6]. In this paper, as a first step, we will consider the common EM scenario that matches two tables A and B, i.e., finds all tuple pairs ($a \in A, b \in B$) that refer to the same real-world entity.

Types of Blockers: As discussed in the introduction, for large tables A and B we typically perform EM by creating a blocker Q, apply Q to A and B to produce a relatively small set of tuple pairs C, then apply a matcher to pairs in C. Over the past few decades blocking has received much attention. The focus has been on developing different blocker types and scaling up blockers, e.g., [18, 22, 33] (see [7, 13] for surveys).

Many blocker types have been developed. MatchCatcher works with all of them. In what follows we briefly discuss the most important types, as Section 6 experiments with many of them.

Well-known blocker types are attribute equivalence, hash, and sorted neighborhood. *Attribute equivalence (AE)* outputs a pair of tuples if they share the same values of a set of attributes (e.g., blocker Q_1: $a.City = b.City$ in Figure 1.b). *Hash blocking* (also called *key-based blocking*) is a generalization of AE, which outputs a pair of tuples if they share the same hash value, using a pre-specified hash function. For example, blocker Q_2 in Figure 1.c combines the hash blocker $lastword(a.Name) = lastword(b.Name)$ and the AE blocker Q_1. *Sorted neighborhood* outputs a pair of tuples if their hash values (also called *key values*) are within a pre-defined distance.

More complex types of blockers include similarity- and rule-based [6, 8, 18]. *Similarity-based blocking (SIM)* is similar to AE, except that it accounts for dirty values, misspellings, abbreviations, and natural variations by using a predicate involving string similarity measures, such as edit distance, Jaccard, overlap, etc. [36]. Examples include $ed(lastword(a.Name), lastword(b.Name) \leq 2$, a blocker which outputs tuple pairs where the last words of their names have an edit distance of at most 2, and blocker $jaccard(a.title, b.title) \geq 0.4$, which outputs pairs of books whose titles have a Jaccard similarity score of at least 0.4. *Rule-based blocking* is perhaps most general. It outputs a tuple pair satisfying a rule or a set of rules encoding domain heuristics, e.g., blocker Q_3 in Figure 1.d consists of two rules. Such blockers can be viewed as the union of multiple blockers, one per rule.

Other types of blockers include phonetic (e.g., soundex), suffix-array, canopy, etc. (see [6, 14] for an extensive discussion).

Efficient Execution of Blockers: Efficient techniques have been developed to execute the above blocker types, both on a single machine and a cluster of machines (e.g., [8, 18, 22]). To execute hash/AE blocking, we partition the tuples in A and B into *blocks*, such that all tuples in each block share the same hash value, then output only pairs of tuples that are in the same block.

To execute a SIM blocker, e.g., $ed(lastword(a.Name), lastword(b.Name) \leq 2$, we build an index I (e.g., prefix filtering index [36]) on the tuples in A, say. Next, for each tuple $b \in B$, we consult I to identify all tuples $a \in A$ such that the pair (a, b) can possibly satisfy $ed(lastword(a.Name), lastword(b.Name) \leq 2$. We check if (a, b) indeed satisfies this predicate, and if yes, then output the pair. Many efficient string indexing techniques [36] can be used to implement SIM blockers. Recent work [8] has also discussed efficient techniques (e.g., using indexing and MapReduce) to execute rule-based blockers.

Accuracy of Blockers: Blocker accuracy is typically measured using recall, defined as follows:

Definition 2.1. [Blocker recall] Suppose applying blocker Q to two tables A and B produces the output C. Let $M \subseteq A \times B$ be the (unknown) set of true matches between A and B, then $recall(Q) = |M \cap C|/|M|$.

Due to dirty data, misspellings, natural variations, synonyms, missing values, etc., no single blocker type produces the highest recall on all datasets. In fact, on any particular dataset, blockers can vary drastically in recalls (e.g., 2.5-98.2% in our experiments).

Finding a blocker with high recall, however, is critical for many EM applications. Counter-terrorism EM applications often need very high coverage., i.e., finding *all* person descriptions that match, and thus want 100% blocking recall. Similar high-coverage examples arise in fraud detection, e-commerce, law, medicine, insurance, and pharmaceutical industry, among others. EM applications with inherently small numbers of matches naturally do not want the blocker to kill off many of these. Finally, EM applications often compute statistics over the matches (e.g., the percentage of patients attending both hospitals), which can be seriously distorted by blockers with low recall.

Limitations of State of the Art: As a result, the topic of blocker accuracy has received growing attention. Proposed solutions include combining multiple blockers to maximize recall (e.g., [12, 20, 22]), and using a sample of tuple pairs labeled as match/no-match to learn blockers with high recall [2, 8, 18, 25].

While promising, these solutions can still produce blockers with varying recalls, oftentimes falling short of 100%. For example, a recent work [8] shows that extensive manual effort to combine hash blockers achieves only 38.8% and 72.6% recall on two datasets. (Obviously we cannot combine *all* possible blockers as there are too many of them.) Another recent work [18] learns blockers using samples labeled by crowdsourcing, but achieves only 92% recall on a data set. In general, due to the difficulties in obtaining a good sample, sampling flukes, etc., today there is still no guarantee that a blocker learned on a *sample* provably achieves high recall when applied to the original tables.

Since there is still no "fool-proof" method to develop a blocker with high recall, it follows that given a blocker Q (either created manually or learned), it is still highly desirable to know how well Q does recall-wise, and what the possible problems are, so that we can improve it. MatchCatcher helps answer these questions, and thus can be considered *complementary* to the above solutions. For example, Section 6 describes a scenario where after the solution in [8] had been used to learn a blocker, we applied MatchCatcher to this blocker and uncovered multiple problems, which can be addressed to further improve the blocker recall.

Overview of MatchCatcher: As discussed, MatchCatcher addresses the following problem:

Definition 2.2. [Finding killed-off matches] Let C be the output of applying blocker Q to tables A and B. Then $D = A \times B - C$ is the set of all pairs killed off by Q. Help the user quickly find as many true matches as possible in D (without materializing it). Examining these matches helps the user understand how well Q does recall-wise, and what can be done to improve its recall.

Figure 2 shows the architecture of MatchCatcher. Given two tables A and B, the Config Generator examines the two tables to generate a set of configs, each of which is a set of attributes (e.g., $\{Name, City\}$). For each config g, the Top-k SSJs module performs a top-k string similarity join to find the k tuple pairs that (a) have the highest score with respect to the attributes in g, and (b) are killed off by blocker Q. Note that to check Condition

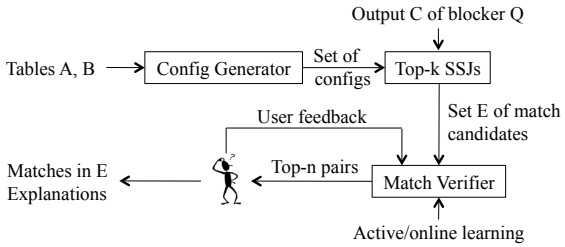

Figure 2: The MatchCatcher architecture

(b), this module does not need to know Q. It only needs to know C, the output of applying Q to A and B. Hence MatchCatcher works independently of the blocker type.

The Top-k SSJs module sends all top-k lists (one per config) to the Match Verifier. This module uses a rank aggregator to combine the lists into a single global list, shows the top n pairs to user U, asks U to identify true matches, uses this feedback together with active and online learning to rerank the pairs, and so on, until U is satisfied or a stopping condition is met. U can examine the true matches to understand how well blocker Q does recall-wise, and to obtain explanations for why these matches are killed off. This helps U decide if Q should be revised, and if so, then how. The next few sections describe MatchCatcher in detail.

3 GENERATION OF CONFIGURATIONS

We now describe the Config Generator, which outputs a set of configs, each being a set of attributes. We cannot consider all possible configs, so the key challenge is to select a good subset of configs. We show how to do so, by carefully managing attributes with many missing values, few unique values, or long string values.

3.1 How Configurations Are Used

We first motivate the notion of configurations (or "configs" for short) and explain how they are being used. Later we build on this to discuss how to select a good set of configs.

Recall that we want to quickly search $D = A \times B - C$, the set of tuple pairs killed off by blocker Q, to find pairs that can be matches. This raises three problems. First, D is not materialized, we only have A, B, and C. Second, even if D is materialized, it would be too large to search quickly. Finally, we do not even know what to search for, since at this point MatchCatcher does not know what a match is (only the user knows).

To address these problems, we begin by assuming that tables A and B share the same schema S (extending MatchCatcher to the case of different schemas is future work). We observe that matching tuples tend to share similar values in a set of attributes, say g (e.g., $\{Name, City\}$). So we want to quickly find tuple pairs in D that share similar values for g and return those as possible matches.

To do so, we convert each tuple a in A into a string $str_g(a)$ that concatenates the values of all attributes in g. For example, if a is (David Smith, Atlanta, 43) and $g = \{Name, City\}$, then $str_g(a)$ is "David Smith Atlanta". This converts Table A into a set A_g of such strings. We convert Table B into a set B_g of strings similarly.

Let $h(x, y)$ be a string similarity measure which computes a score in $[0, 1]$ between two strings x and y. Examples of such measures are Jaccard, cosine, overlap, edit distance, etc. [36]. Then next we perform a top-k string similarity join (SSJ) between A_g and B_g to find the k tuple pairs in $A \times B$ with the highest $h(x, y)$ score. Techniques have been developed to quickly perform top-k SSJs [34, 37]. Of course, our goal is not to find pairs in $A \times B$, but rather in $D = A \times B - C$. We can modify the above techniques slightly to ensure this, by dropping a found pair if it is in C. We then return the k pairs in D with the highest $h(x, y)$ score as possible matches.

The above procedure does not require a materialized D, only tables A, B, and C (the output of blocker Q). It can quickly search D using a modified version of top-k SSJs to return possible matches. Of course, at this point we still do not know if these are indeed matches. But later we can work with the Match Verifier to quickly shift through them to find true matches, if any. We now discuss several important aspects of the above procedure.

Why Concatenating the Attributes? We can use a variety of methods to find tuples that share similar values for attributes in g, e.g., finding pairs that share similar values for *each* attribute in g, then taking their intersection, say. However, given the interactive nature of debugging, we want this step to be as fast as possible. Hence we decide not to treat the attributes in g separately, but concatenate all of them into a single string, then compare them using SSJs. Section 4 shows that this method can quickly search a very large set D. But a drawback is that we can return *false positives* such as tuple pair (Jim Madison, Smithville, 32) and (Jim Smith, Madison, 32), because their concatenated strings are very similar given certain similarity measures. Such false positives, however, can be "weeded out" in the Match Verifier, using user feedback and active/online learning (see Section 5).

Which String Similarity Measure to Use? Given that similar attribute values can still vary significantly (e.g., "Dave Smith" vs "David Frederic Smith"), measures that treat strings as sets (e.g., Jaccard, cosine, etc.) typically work better than those that treat strings as sequences of characters (e.g., edit distance) [9]. So for MatchCatcher, we use the well-known Jaccard measure that tokenizes two strings x and y into two sets of words P_x and P_y, then returns $|P_x \cap P_y|/|P_x \cup P_y|$ [34]. However, Theorem 4.2 shows that our solution can also work with other set-based similarity measures, namely overlap, cosine, and Dice [34].

Why Multiple Configurations? So far we have used just one config g to find match candidates. Using multiple configs, however, can produce more matches. For example, config $\{Name, City\}$ may not return the pair (David Smith, Seattle) and (Dave Smith, Redmond) because the cities are different. Config $\{Name\}$ however can. Conversely, config $\{Name\}$ may not return the pair (Chuck Smith, San Francisco) and (Charles F. Smith, San Francisco) because the names are too different, but config $\{Name, City\}$ can. Together, these two configs can return more matches than either of them in isolation. Generating a good set of configs however is a major challenge, which we address next.

3.2 Generating Multiple Configurations

As a baseline, we can use all subsets of attributes in S (the schema of A and B) as configs. But this generates too many configs even for a moderate size (e.g., $|S| = 8$ produces $2^{|S|} - 1 = 255$ configs). We cannot use all of them because the total SSJ time would be too high. So we must select a smaller set of configs.

To do so, we select a set of promising attributes in S, then use them to generate configs, in a top-down fashion. In each step of the process, we select which configs to generate next by carefully considering the impact of attributes with many missing values, few unique values, or long string values. The end result is a *config tree* consisting of multiple configs. Later the Top-k SSJs module

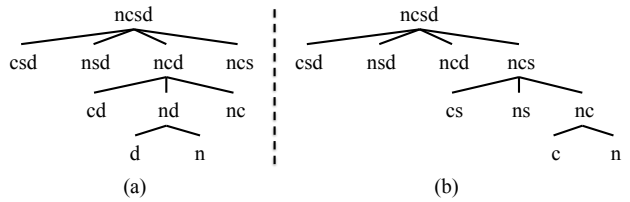

Figure 3: An example of generating config trees.

will traverse this tree to perform top-k SSJs on the configs in a joint fashion. We now elaborate on these steps.

Selecting the Most Promising Attributes: We first classify attributes in S as string, numeric, categorical, and boolean, using a rule-based classifier. Next, we drop numeric attributes (e.g., Salary, Price) because matching tuples still often differ in their values (e.g., the same product having different prices). Finally, we drop categorical and boolean attributes whose appearances in tables A and B are different. For example, if Gender has values $\{Male, Female\}$ in A but $\{M, F, U\}$ in B, then we drop it as these two sets share no value (in general if the Jaccard score of these two sets is less than a pre-specified threshold then we drop the attribute). The remaining attributes are string, or categorical/boolean but with similar sets of values. We return these as T, the set of the most promising attributes to be used for config generation. (Of course, the user can also manually curate schema S to generate T. The experiments in Section 6 however do not involve any manual curation.)

Generating a Config Tree: Given the set T of promising attributes, we generate a *config tree* in a top-down fashion, then return all configs in the tree. Specifically, we start with T as the config at the root of the tree. Next, we "expand" this node by removing each attribute from T to obtain a smaller config of size $|T| - 1$. This produces $|T|$ new configs, which form the nodes at the *next level* of the tree. We then select just one node at this level to "expand" further, and so on (we will discuss how to select shortly). This continues until we have reached configs of just one node. Figure 3.a shows an example config tree, assuming $T = \{n, c, s, d\}$ (which stand for Name, City, State, and Description, respectively).

Intuitively, this strategy ensures that we generate a diverse set of $|T|(|T|+1)/2$ configs of varying size $|T|, |T|-1, \ldots, 1$. The config tree will also be used to guide the joint execution of top-k SSJs on the configs (see Section 4.2). We now turn to the challenge of how to select a node to expand in the config tree.

Managing Many Missing Values and Few Unique Values: Consider again the config tree in Figure 3.a. Suppose we are currently at the second level of the tree, and need to select one node among the four nodes *csd*, *nsd*, *ncd*, and *ncs*, to expand. This selection is equivalent to *selecting an attribute to exclude from subsequent config generation.* Indeed, if we exclude attribute *s*, then we select node *ncd* to expand (as shown in the figure). Otherwise if we exclude *d*, then we select the rightmost node *ncs* to expand, and so on.

So which attribute should we exclude? We observe that if an attribute has many missing values, then keeping it for subsequent config generation is not desirable, because we will end up with configs that produce similar top-k lists. For example, suppose we have selected config *ncd* to expand (as shown in Figure 3.a), and suppose that *d* has many missing values, then many strings for config *ncd* and config *nc* will be identical, potentially leading to similar top-k lists. In the extreme case, if all values for *d* are missing, then these two top-k lists are identical. Clearly, we want different configs to produce substantially different top-k lists, to avoid redundant work and to maximize the number of matches we can retrieve.

Name: Bryan Lee, **City**: Austin, **State**: TX,
Desc: Joined in 8/2003, promoted to team lead 5/2005, promoted to director of sales 4/2009. Currently on unpaid leave until 1/2013.

Name: Bryan M. Lee, **City**: Austin, **State**: TX,
Desc: Outstanding customer service record 03-05. Achieved sales of $2M/year 05-09. Shortlisted for VP of sales 2011. Shortlisted for VP of marketing 2012.

Figure 4: Examples of tuples with long string attributes.

Another observation is that if an attribute has more unique values than another, e.g., *c* vs *s* (which stand for City and State, respectively), then it is better to exclude *s*, the one with fewer unique values, because intuitively, if two tuples agree on *c*, they are more likely to match than if they agree on *s*, all else being equal. Thus, to maximize the number of matches we can retrieve, we should strive to keep the "more specific" attributes, i.e., the ones with more unique values.

Combining the above two observations, we define the e-score (shorthand for "expected benefit") of an attribute as follows:

Definition 3.1. [E-score of an attribute] Let $n_A(f)$ be the ratio of the number of non-missing values of attribute f in A over the number of tuples in A, and $u_A(f)$ be the ratio of number of unique values of f in A over the number of non-missing values of f in A. We define $n_B(f)$ and $u_B(f)$ similarly. Define $e_A(f) = 2n_A(f)u_A(f)/[n_A(f) + u_A(f)]$ and define $e_B(f)$ similarly. Then we define the e-score of attribute f as $e(f) = e_A(f)e_B(f)$.

We then select the attribute with the lowest e-score to exclude at each level of the config tree. For example, suppose $e(n) > e(d) > e(c) > e(s)$. Then at the second level of the tree in Figure 3.a, we exclude attribute *s*, which means selecting node *ncd* to expand. At the third level of the tree, we exclude *c*, which means selecting node *nd* to expand.

Managing Long String Attributes: Many datasets contain attributes with long string values, e.g., Comment, Desc, etc. Figure 4 shows two tuples where attribute Desc has such long values. Such long attributes can cause two problems. First, they can cause multiple configs to generate very similar top-k lists.

Example 3.2. Consider again the config tree in Figure 3.a. Suppose attribute d has long string values (such as those shown in Figure 4). Then all seven configs involving d can generate similar top-k lists because the long values of d "overwhelm" the short values of the remaining attributes. So when moving from a config involving d to another (e.g., from ncd to nd), the strings do not change much, and therefore their similarity scores also do not change much (we formalize this notion below), leading to similar top-k lists.

The second problem is that if the long string values are different for matching tuples, then a config involving this long attribute will fail to return the match. For example, the two tuples in Figure 4 match, but any config involving attribute Desc will not return this match, because the values for Desc here are very different, and so the score between the two tuples will be low.

To address this, we modify our config-tree generation procedure as follows. Suppose we need to select a config node in the tree to expand. Before, we select $g_{default}$, the one without the attribute with the smallest e-score. Now, we first run a procedure FindLongAttr to see if there is any attribute that is "too long" (i.e., likely to adversely affect selecting good configs). If such an

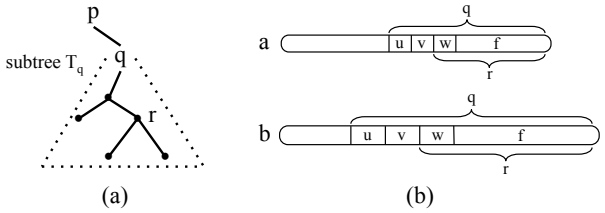

Figure 5: Finding attributes judged too long.

attribute f_{long} exists, then we select the config without f_{long} to expand. Otherwise we select $g_{default}$, as usual.

Example 3.3. Consider again Figure 3.a, which shows the "default" config tree with root ncsd. To handle long attributes, once we are at the second level, we do not automatically select ncd (the config without s, the attribute with the smallest e-score) for expansion. Instead, we run FindLongAttr *at this level. Suppose it returns d (thus judging d to be too long). Then we select ncs, the config without d, for expansion. This produces new configs cs, ns, and nc (see Figure 3.b). Suppose running* FindLongAttr *at the level of these new configs returns no attribute. Then we select config nc (the config without s, the attribute with the smallest e-score) for expansion (see Figure 3.b).*

We now explain procedure FindLongAttr. The key challenge is to formalize what it means to be "too long". Let p be a node in the config tree. Suppose that when running the default config generation procedure (the one that does not consider long attributes), we end up selecting q, a child node of p, for expansion, and that we eventually generate a subtree T_q rooted at q (see Figure 5.a).

We say that an attribute f is too long if it "overwhelms" many config nodes in subtree T_q, specifically if it overwhelms at least half of the configs in $F(T_q)$, the set of configs in T_q that contain f. In turn, we say that f overwhelms a config $r \in F(T_q)$ (see Figure 5.a) *if the top-k list obtained from config r is "roughly the same" as the top-k list obtained from config q (we formalize this below).* Intuitively, we want to avoid such cases, because we want each config to return a different top-k list, to maximize the number of true matches that we will find. So if we find that f overwhelms at least half of the configs in $F(T_q)$, then we judge f to be too long and should be removed. That is, instead of selecting q for expansion, we will select the config (in the same tree level as q) that does not contain f.

Of course, we do not have access to the top-k lists of r and q. So we develop a condition which if true would suggest that the two lists are "roughly the same". Specifically, let $sim_g(a,b) = h(str_g(a), str_g(b))$ be the string similarity function between the string values of two tuples a and b, for config g. Suppose that for all tuple pairs (a,b) in $D = A \times B - C$ we have

$$Condition\ 1: |sim_q(a,b) - sim_r(a,b)|/sim_q(a,b) \leq \alpha,$$

for a small pre-specified α value, say 0.2. Then we can say that when we switch config from q to r, the score of each tuple pair does not change much, so the top-k list for r will stay roughly the same as that of q.

Checking Condition 1 for all pairs (a,b) in D is not feasible. Hence we perform a theoretical analysis for an idealized scenario (described below). Of course, this idealized scenario rarely happens in practice. But understanding it helps us come up with an efficient heuristic to check Condition 1.

Let $L_f(a)$ be the length (i.e., the total number of words) of attribute f in tuple a, $L_q(a)$ be the sum of the lengths of all attributes in q, for tuple a, and so on. The idealized scenario assumes that (a) attribute f takes the same proportion of the total length of q in both a and b, i.e., $L_f(a)/L_q(a) = L_f(b)/L_q(b) = \beta$, and (b) the remaining length of q is equally distributed among the remaining attributes of q, i.e., $L_k(a) = [(1-\beta)L_q(a)]/(|q|-1)$ for all attribute k in $q - \{f\}$, and the same condition applies to tuple b.

Example 3.4. Consider the two tuples a and b in Figure 5.b, where $q = \{u, v, w, f\}$ and $r = \{w, f\}$. We assume that $L_f(a)/L_q(a) = L_f(b)/L_q(b) = \beta$, and $L_u(a)/L_q(a) = L_v(a)/L_q(a)$ and $L_u(b)/L_q(b) = L_v(b)/L_q(b)$.

Then we can show that (see [24] for a proof sketch):

Theorem 3.5. *Let $a \in A$ and $b \in B$ be two tuples that satisfy the above assumptions. If*

- *(R_1) $sim_q(a,b) \geq [\sqrt{(1+\alpha)^2+8} - (1+\alpha)]/4$, and*
- *(R_2) $\beta \geq 1 - \frac{(|q|-1)}{|q \backslash r|} \cdot \frac{\alpha}{(1+\alpha)} \cdot \frac{max\{L_q(a), L_q(b)\}}{L_q(a)+L_q(b)}$,*

then pair (a,b) satisfies Condition 1.

Intuitively, this theorem says that if $sim_q(a,b)$ is sufficiently high (Requirement R_1), and attribute f is sufficiently long (Requirement R_2), then $sim_r(a,b)$ will be close to $sim_q(a,b)$. It is not difficult to show that the quantity on the right-hand side of R_1 is upper bounded by 0.5. In practice, we observe that users typically examine only the *top few tens of pairs* in each top-k list (see Section 5), and that if these pairs are true matches, their scores often exceed 0.5, making R_1 true. As a result, if R_2 is also true, then attribute f is long enough to "overwhelm" these pairs. That is, these pairs will change little score-wise when switching from config q to r, thus typically will still show up in the top few tens of pairs of the top-k list for r, an undesirable situation.

To avoid such situations, we will focus on checking R_2. Checking R_2 for many pairs (a,b) is not practical. So we approximate this checking using average lengths, i.e., we (a) replace β in the left-hand side of R_2 with $min\{AL_f(A)/AL_q(A), AL_f(B)/AL_q(B)\}$, where $AL_f(A)$ for example is the average length of attribute f in Table A, and (b) replace $L_q(a)$ and $L_q(b)$ in the right-hand side of R_2 with $AL_q(A)$ and $AL_q(B)$, respectively.

Procedure FindLongAttr then works as follows. Suppose we have selected config q for expansion (because it does not contain s, the attribute with the least e-score). Then for each attribute f (other than s), we (a) identify $F(T_q)$, the set of configs in T_q that contain f, (b) declare f "too long" if the above approximate checking is true for at least half of the configs $r \in F(T_q)$. It is not difficult to prove that at most one attribute f will be found too long. If so, we do not select q, but select instead the config that does not contain f for expansion. Otherwise, we select q, as usual. This procedure takes less than a second in our experiments.

Discussion: Note that we do not completely remove attributes with many missing values, few unique values, or long values from config generation. Instead, each such attribute f may be removed *only at some point* during the generation process. Configs generated earlier still contain f.

Further, our work here is related to, but very different from work such as [3, 10]. Those works find attributes that are discriminative for classification, often using a labeled sample (as many works in learning do). Here we do not look for discriminative attributes. Instead, we look for attributes such that if two tuples agree on their values, then they are likely to match. For example, suppose all tuples in table A have the same value "US" for "Country", and all tuples in table B have the same value "Canada".

Then "Country" is a discriminative attribute because if two tuples disagree on it, they definitely do not match. For our purpose, however, "Country" has little expected benefits, because if two tuples agree on it, it is still not likely that they match (not as much as if they agree on "State" and "City" say).

In fact, the work [29] also treats attributes with missing values and few unique values in a way similar to ours (for blocking and matching). However, it does not handle long attributes, and uses only one config, and thus is significantly outperformed by MatchCatcher (see Section 6).

4 TOP-K STRING SIMILARITY JOINS

So far we have discussed generating a set of configs. We now discuss performing top-k SSJs over these configs (one per config). Previous work has discussed top-k SSJs for a single config [34]. Here we significantly improve that work (and our solution can be applied to top-k SSJ situations beyond this paper). We then discuss executing multiple top-k SSJs jointly, by reusing results across the configs, in a parallel fashion.

MatchCatcher currently works with the Jaccard string similarity measure, and we will explain it using that measure. However, it is important to note that all algorithms discussed below also work with the set-based similarity measures cosine, overlap, and Dice.

4.1 Improving Top-k Join for a Single Config

As far as we can tell, the state of the art in top-k SSJs is TopKJoin [34]. Given a set J of strings, TopKJoin finds the k string pairs with the highest similarity scores, for a pre-specified k, in a branch-and-bound fashion. Specifically, it maintains a prefix for each string in J, incrementally extends these prefixes, finds string pairs whose prefixes overlap, computes their similarity scores, use these scores to maintain a top-k list, then extends the prefixes again, and so on.

Example 4.1. Suppose J consists of the four strings w, x, y, z in Figure 6.a. We begin by creating a prefix $p(w)$ = "a" for w, then a prefix $p(x)$ = "a" for x. At this point the prefixes of the pair (x, w) overlap. Hence we compute the Jaccard score 0.8 for this pair, then initializes the top-k list K to be containing just this pair. (Here we assume $k = 2$.)

Next, we create prefix $p(y)$ = "b". This does not produce any new pair whose prefixes overlap. So we continue by creating prefix $p(z)$ = "b". This produces a new pair whose prefixes overlap: (z, y) with score 0.43. Figure 6.b shows the updated top-k list K.

Next, we select one prefix to extend (we will discuss shortly how). Suppose we select $p(x)$ and extend it by one token. Then $p(x)$ = "ab" (see Figure 6.a). This produces two new pairs whose prefixes overlap: (x, y) with score 0.67 and (x, z) with score 0.43. Figure 6.c shows the updated top-k list K. We then select another prefix to extend, and so on. Finding new pairs with overlapping prefix can be done efficiently using an inverted index from token to the prefixes of the strings [34].

We now discuss how to select a prefix to extend. Suppose we have imposed a global ordering on all tokens, and sorted the tokens in each string w, x, y, z in that order (see Figure 6.a). Suppose also that we have created prefixes of size 1, namely $p(w)$ = "a", $p(x)$ = "a", $p(y)$ = "b", $p(z)$ = "b", and are now deciding which prefix to extend. Suppose we select $p(w)$ and extend it by one token, to be "ab". Then it is easy to show that the scores of all *new* pairs generated by this extension are capped by 0.75. Indeed, any *new* pair must involve w. Let such a pair

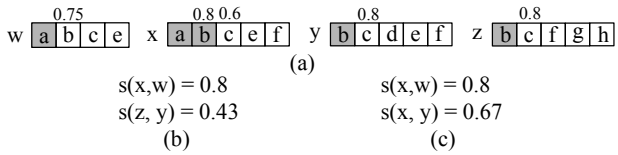

Figure 6: An illustration of top-k computation.

be (w, v). Then the first common token that they share should be "b" (the token just being added to $p(w)$). So they can share at most this token b and the remaining "unseen" tokens of w. Thus $|w \cap v| \leq 3$. Since $|w \cup v| \leq |w| = 4$, it follows that the Jaccard score of (w, v) is capped by 3/4=0.75. We write 0.75 on top of token "b" in w to indicate that when we extend $p(w)$ to include this token, the score of any *new* pair generated by TopKJoin will be *capped* by 0.75. Similarly, we can write 0.8 for the second tokens of x, y, z (see Figure 6.a).

We then select the prefix that when extended will include the token with the highest "cap" number (in the hope that it will generate new pairs with the highest possible scores). In this case, we select $p(x)$ (but $p(y)$ and $p(z)$ also work).

We now discuss how to stop. Observe that the "cap" number for "c" in x is 0.6. By the time we have to consider whether to extend $p(x)$ to include "c", the top-k list already has a lower-bound score of 0.67 (see Figure 6.c), greater than 0.6. As a result, we do not have to extend $p(x)$ to include "c", and in fact, prefix extension on x can be stopped at this point. TopKJoin terminates when all prefix extensions have stopped, either early (as described above) or because the prefix has covered the entire string. The paper [34] describes TopKJoin in detail, including optimizations to avoid redundant computations.

The QJoin Algorithm: TopKJoin has a major limitation. Every time it generates a new pair (u, v), it immediately computes the similarity score of (u, v) (then updates the top-k list). Computing this score turns out to be very expensive, especially if u and v are long strings. In a sense, it is also "premature", because it can be shown that when we generate (u, v) (as a new pair), we only know that they share a *single* token. There is no evidence yet that they share more tokens and thus are likely to have high similarity score. If they indeed share only one or few tokens, and yet we still compute their score, then that score is likely to be low. So the pair will not make it into the top-k list, yet we have wasted time computing it score.

To address this problem, when generating new pairs, we do not immediately compute their scores. Instead, we keep track of the number of common tokens each pair has, and update this number whenever a prefix is extended. We then compute the score of a pair only if it has q common tokens, and thus is likely to have a high score. It is difficult to select q analytically, so we select it empirically as follows. Assuming at least four CPU cores, we begin by running the modified TopKJoin for $q = 1$, $q = 2$, etc., on all cores, one q value for each core, for $k = 50$. (Note that TopKJoin always does $q = 1$.) Then whichever core finishes first, we keep the process on that core running to produce the rest of the top-k list (effectively selecting the q value associated with that core), and kill the processes on the other cores.

It is straightforward to adapt the above algorithm to work with two tables (instead of just one), and to remove a pair from the top-k list (during the top-k computation) if it happens to be in the candidate set C. Henceforth we refer to this new algorithm as QJoin.

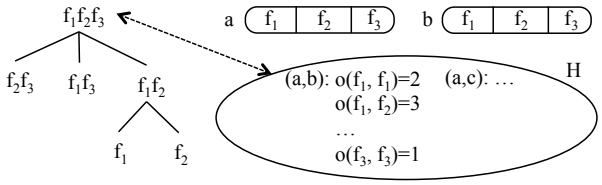

Figure 7: Reusing across top-k computations.

4.2 Joint Top-k Joins Across All Configs

TopKJoin can only be applied to a single config [34]. Our setting however involves multiple related configs. We now describe a solution to find top-k lists *jointly across the configs*. To do so, we use QJoin, but modify it to reuse similarity score computations and top-k lists across the configs, and process the configs in parallel.

Reusing Similarity Score Computations: As discussed in Section 4.1, computing the similarity score of a pair (a, b) is very expensive, especially for long strings. Hence, we want to reuse such computations across the configs. To do so, we process the configs in the config tree in a breadth-first order, e.g., processing the root config $f_1f_2f_3$ of the config tree in Figure 7 (where the f_i-s are attributes), then the next-level configs, f_2f_3, f_1f_3, f_1f_2, and so on.

When processing a config g (i.e., finding its top-k list), we keep track of certain information, then reuse it when processing configs in the subtree of g. For example, consider again the config tree in Figure 7. We start by tokenizing the strings wrt the root config $f_1f_2f_3$ into multisets of word-level tokens. Next, we process the config $f_1f_2f_3$. This process computes the Jaccard score of multiple tuple pairs. When computing the score of such a pair, say (a, b), we compute and store the number of overlapping tokens between any two attributes f_i of a and f_j of b in an in-memory database H. Figure 7 illustrates this step. Here, $o(f_1, f_1) = 2$ means attributes f_1 of a and f_1 of b share two tokens. (We only store in H attribute pairs that share tokens.)

Then we can reuse H to drastically speed up processing configs in the subtree rooted at $f_1f_2f_3$. For example, consider processing config f_1f_2. If during this process we need to re-compute the score of (a, b) (now with respect to only f_1 and f_2), then we can use H to compute $Overlap_{f_1f_2}(a, b) = o(f_1, f_1) + o(f_1, f_2) + o(f_2, f_1) + o(f_2, f_2)$, then compute the above score as

$$Overlap_{f_1f_2}(a, b)/(L_{f_1f_2}(a) + L_{f_1f_2}(b) - Overlap_{f_1f_2}(a, b)),$$

where $L_{f_1f_2}(a)$ for instance is the length in tokens of the concatenation of f_1 and f_2 for a. Computing the score of (a, b) this way is far faster than computing from scratch.

Note that while processing config f_1f_2, if we have to compute the score of a new pair (c, d) not yet in H, then we will store similar overlap information for (c, d) in H, to enable reuse when processing configs in the subtree rooted at f_1f_2, and so on.

Reusing Top-k Lists: When applying to a config g, algorithm QJoin starts with an empty top-k list K, then gradually grows K as it iteratively expands the prefixes. In our setting, however, since we process multiple configs, a promising idea is to use the top-k list of a previous config to initialize the top-k list of the current config.

For example, after processing config $g = f_1f_2f_3$ (Figure 7), we store its top-k list K_g. Then when processing config $h = f_1f_2$, we use the database H described earlier (which stores overlap information) to re-adjust all scores in K_g. This is necessary because these scores are computed wrt $f_1f_2f_3$, but now we want them to be adjusted to consider only f_1f_2. This re-adjustment is fairly straightforward (and inexpensive) because the overlap information for all pairs in K_g should already be in H. Next, we run the algorithm QJoin as usual to process config $h = f_1f_2$, but using the K_g list with the adjusted scores as the initial top-k list K_h (instead of using an empty list).

Observe that the above procedure enables reusing top-k lists from a parent to a direct child (e.g., from $f_1f_2f_3$ to f_1f_2). Reusing across the "siblings" appears much more difficult. For example, given the top-k list for f_1f_3, there is no obvious way to quickly adjust its scores for f_1f_2, using database H. Hence, currently we do not yet consider such sibling reuse.

Finally, reuse does not come for free. It helps avoid computing certain similarity scores from scratch, but incurs an overhead of storing and looking up the overlap information. If the tuples are short, then the overhead can easily overwhelm the savings. As a result, we trigger reuse only if the average tuple length is at least t tokens (currently set to 20).

Parallel Processing of the Configs: Finally we explore parallel processing on multiple cores. (We consider multicore single machines for now because it is a common setting for many domain science users [19].) An obvious idea is to process each config across multiple cores. For example, we can split Table A into two halves A_1 and A_2 and Table B into B_1 and B_2, find the top-k list for A_1 and B_1 on the first core, the top-k list for A_1 and B_2 on the second core, etc., then merge the top-k lists. In practice, this approach suffers from severe skew: one core finishes quickly while another runs forever. While it is possible to split the tables intelligently to mitigate skew, this adds considerable overhead and implementation complexity.

As a result, we opted for processing one config per core. Specifically, we traverse the config tree breadth-first, and assign configs to cores in that order (when a core finishes, it gets the next config "in queue"). This solution *continuously utilizes all cores*. But it raises two problems. First, two configs (e.g., $f_1f_2f_3$ and f_1f_2) may concurrently write, or one reads and the other writes, into database H, causing concurrency control issues. To address concurrent writes, observe that only configs with non-empty subtrees (e.g., $f_1f_2f_3$ and f_1f_2 in Figure 7) will write. For each such config g, we require it to write into a separate in-memory database H_g.

To address dirty reads (e.g., $f_1f_2f_3$ writes into a database while f_2f_3 reads from it), we note that here each "write" just *inserts* a value; it never modifies or deletes. For such cases there are atomic hashmaps that perform *atomic inserts*, thus avoiding dirty reads. So we implement each database H_g as one such hashmap (using the Atomic Unordered Hashmap in Facebook's C++ Folly package).

Finally, if a parent config, e.g., $g = f_1f_2f_3$, has not yet finished, then a direct-child config, $h = f_1f_2$, cannot reuse g's top-k list. In such situations, we start config h with an initial empty top-k list. When config g finishes, it sends its top-k list to h. Config h merges its current top-k list with the new top-k list from g, to obtain a potentially better top-k list, then continues. The technical report [24] shows the pseudo code of the complete algorithm, and the following theorem shows its correctness (see [24] for a proof sketch):

THEOREM 4.2. *Given two tables A and B, the output C of a blocker on A and B, a set of configs G, a string similarity measure which is Jaccard, consine, overlap, or Dice, and a value k, the above algorithm returns a set of top-k lists, where each top-k list is the*

output of applying Algorithm QJoin *to A, B, and C, using a config $g \in G$ and the given similarity measure and k value.*

5 INTERACTIVE VERIFICATION

So far we have discussed processing configs to obtain a set of tuple pairs. We now discuss identifying true matches in this set, via user engagement, rank aggregation, and active/online learning.

Engaging the User: Let E be the union of the top-k lists obtained from processing all configs. Typically E is large (e.g., 3,011-7,089 in our experiments) and the true matches make up just a small portion of E. Thus expecting a user U to be able to examine the entire set E to find true matches is unrealistic.

A reasonable solution is to rank the pairs in E such that the true matches "bubble" to the top, then present the ranked list to U. However, our experiments with a variety of ranking methods (see below) suggest it is very difficult to do so. Typically, the top of the ranked list indeed contains multiple matches. But then the remaining matches tend to be scattered far and wide in the list.

As a result, we decided to engage user U: we rank the pairs in E, present the top-n pairs to U (currently $n = 20$), ask U to identify the true matches, use this feedback to rerank the list, then present the next top-n pairs to U, and so on. As such, we help U iteratively identify true matches, but use this identification to help "bubble" the remaining matches to the top of the ranking.

Using Rank Aggregation: Let m be the number of configs and $L_1, \ldots, L_m$ be the top-k lists obtained from these configs. To engage user U, we first need to aggregate these lists into a single list. Many aggregation methods exist, e.g., [4, 15]. Here we use MedRank [15], a popular method. To use MedRank, we first sort each list L_i in decreasing order of score, then associate each item in the list with a rank, i.e., an integer, such that the higher the score, the lower the rank and items with the same score receive the same rank. Next, we compute for each item a global rank which is the median of its ranks in the lists. Finally, we sort the items in increasing order of global rank, breaking ties randomly, to obtain a list $L*$ which is the aggregation of all top-k lists L_i-s.

Example 5.1. Figure 8 shows three top-k lists L_1, L_2, L_3 and the global list $L*$. A line such as "a: 1.0 (1)" under L_1 means that item "a" in list L_1 has score 1.0 and has been assigned rank 1. The ranks for "a" is 1, 1, 2 (see Figure 8). So its global rank is 1. The ranks for "b" is 2, 4, 1 (here "b" is missing from L_2, which has ranks 1-3; so we assign to it rank 4). Thus "b"'s global rank is 2. And so on.

Once we have obtained the global list $L*$, we can present the top-n items of $L*$ to user U. But how do we incorporate the user feedback for the next iteration? A reasonable solution is to use weighted median ranking (WMR): we first assign an equal weight $w_i = 1/m$ to each top-k list L_i ($i \in [1, m]$). At the end of the first iteration, we adjust $w_i = w_i \cdot [1 + log(1 + r_i)]$, where r_i is the number of true matches user U has identified that appear in L_i, then normalize all weights w_i. At the start of the next iteration, we merge the lists $L_1, \ldots, L_m$ again, using WMR to compute the global rank of each item. Next, we present the top-n pairs in this merged list to the user, and so on. Intuitively, the top-k lists in which more true matches appear will become more important, and the weighted global ranking will be "leaning toward" those lists.

Using Learning: WMR does not perform well in our experiments (see Section 6). It uses a very limited combination model which fails to fully utilize user feedback. To address this, we explored active learning. Specifically, we iteratively show the next n items of $L*$ to user U, until we have obtained at least one match and one non-match. Suppose we have carried out t iterations, then this produces a set T of nt labeled items. We use T to train a random forest classifier F, use F to find n most informative items in $L*$, show them to the user to label, add the newly labeled items to T, then retrain F, and so on.

L_1	L_2	L_3	$L*$
a: 1.0 (1)	a: 0.9 (1)	b: 0.8 (1)	a (1)
b: 0.8 (2)	c: 0.7 (2)	a: 0.5 (2)	b (2)
c: 0.8 (2)	d: 0.6 (3)	c: 0.3 (3)	c (2)
d: 0.6 (4)		d: 0.2 (4)	d (4)

Figure 8: Combining top-k lists using MedRank.

Active learning alone however is not quite suited for our purpose. Its goal is to learn a good classifier as soon as possible. Hence it typically shows user U controversial items that it finds difficult to classify. But many or most of these items can be non-match. User U, however, wants to find many *true matches* as soon as possible (so that U can examine them to quickly understand the problems with the blocker).

The above two goals conflict. To address this problem, we adopt a hybrid solution. After we have obtained the training set T and trained a classifier F, as described above, for the next iteration, we show user U n items where $n/4$ items are the top controversial items chosen by F, as described above. The remaining $3n/4$ items however are those with the *highest positive prediction confidence*, where the confidence is computed as the fraction of decision trees in F that predict the item as a match. Intuitively, the first $n/4$ items are intended to help the active learner, whereas the remaining $3n/4$ items can contain many true matches, and are intended to help the user quickly find many true matches in the first few iterations.

After three such iterations, we stop active learning completely (judging that classifier F has received enough labeled controversial examples in order to do well), but continue the online-learning process with F. Specifically, in each subsequent iteration, we show user U the top n items with the highest positive prediction confidence, produced by F. Once these items have been labeled by U, we add them to the existing training set, retrain F, and so on.

When to Stop? A natural stopping point is when user U finds no new matches in 2 consecutive iterations. Of course, U can stop earlier or continue. If the required blocker recall is very high, U can continue for many iterations. Otherwise, U can stops after the first few iterations (because if these iterations contain many matches, then examining them often already reveals problems with the blocker, which U can then fix).

6 EMPIRICAL EVALUATION

We evaluated MatchCatcher in three ways. First, we experimented with a broad range of blockers that vary in recall, types, and complexity, representing blockers that users may write *at various points* during the blocker development process. We show that MatchCatcher works well with these blockers, thus can effectively support the users in the development process.

Second, we experimented with blockers that are either the best hash blockers manually developed or the best blockers automatically learned using a state-of-the-art solution. We show that even in these cases MatchCatcher can help uncover problems and improve the blockers.

Dataset	Tuple type	Table A	Table B	# of matches	# of attrs	Average length
Amazon-Google	software product	1363	3226	1300	5	205, 38
Walmart-Amazon	electronic product	2554	22074	1154	7	76, 179
ACM-DBLP	paper	2294	2616	2224	5	16, 19
Fodors-Zagats	restaurant	533	331	112	7	11, 10
$Music_1$	song	100000	100000	2978	8	9, 9
$Music_2$	song	500000	500000	73646	8	9, 9
Papers	paper	455996	628231	unknown	7	17, 18

Table 1: Datasets for our experiments.

Dataset	Blocker Q
A-G	**(OL)** title_overlap_word<3 **(HASH)** attr_equal_manuf **(SIM)** title_cos_word<0.4 **(R)** title_jac_word<0.2 AND manuf_jac_3gram<0.4
W-A	**(OL)** title_overlap_word<3 **(HASH)** attr_equal_brand **(SIM)** title_cos_word<0.4 **(R)** price_absdiff>20 OR title_jac_word<0.5
A-D	**(OL)** authors_overlap_word<2 **(SIM)** title_jac_3gram<0.7 **(R_1)** title_cos_word<0.8 AND authors_jac_3gram<0.8 **(R_2)** year_abs_diff>0.5 OR title_jac_word<0.7
F-Z	**(OL)** name_overlap_word<2 **(HASH)** attr_equal_city **(SIM)** addr_jac_3gram<0.3 **(R)** (name_cos_word<0.5 AND type_jac_3gram<0.7) OR addr_jac_3gram<0.3
M_1	**(OL)** artist_name_overlap_word<2 **(HASH)** attr_equal_artist_name **(SIM)** title_cos_word<0.5 **(R)** year_absdiff>0.5 OR title_cos_word<0.7
M_2	**($HASH_1$)** attr_equal_artist_name **($HASH_2$)** attr_equal_release_OR_attr_equal_artist_name **(SIM_1)** title_cos_word<0.6 **(SIM_2)** title_cos_word<0.7 **(SIM_3)** title_cos_word<0.8

Table 2: Blockers for the first set of experiments.

Finally, we asked real-world users in several data science classes, domain science projects, and at several organizations to use MatchCatcher. We show that MatchCatcher has proven highly effective in helping these users develop blockers.

6.1 Supporting Users in Developing Blockers

For this experiment we need "gold" matches, so we use the six datasets shown in the first six rows of Table 1. As far as we can tell, these datasets are the largest ones used in previous EM work for which "gold" matches are available. Here we created two versions of the Music dataset, Music1 and Music2, to ensure a diversity of size (from 331 to 100K to 500K of tuples per table). The technical report [24] describes these datasets in details.

For each dataset we asked volunteers to create multiple blockers (see Table 2). They are of the types described in Section 2: overlap (OL), hash (HASH), similarity-based (SIM), and rule-based (R). For example, the first row of Table 2 describes 4 blockers for dataset A-G. These include a hash blocker on attribute "manufacturer" and a rule-based blocker that combines two SIM blockers. See [24] for more details on these blockers. (The next subsection describes experiments with the best hash blockers manually created for these datasets.)

Developing a blocker is typically *a long process* in which users often start with a simple blocker, then revise it into more complex ones with higher recall. The above blockers differ in type, recall, and complexity, representing blockers that users may write *at various points* during the above process. We now show that MatchCatcher can help debug these blockers, suggesting that it can support the user during the entire development process.

Overall Accuracy: First we examine the top-k SSJs module. The first two columns of Table 3 list datasets and blockers. Column C lists the size of C, the output of the blocker on Tables A and B. Column M_D lists the number of true matches in $D = A \times B - C$. This number varies drastically, e.g., 137-1,267 for A-G, 87-566 for W-A, etc., suggesting that blocker recall often varies widely and that it is important to debug to improve recall.

Column E lists the size of E, the union of all top-k lists over the configs (for $k = 1000$). Column M_E lists the number of true matches in E (the numbers outside parentheses), and shows that set E contains a substantial fraction of true matches in D, e.g., 54-65% for A-G, 41-83% for W-A, 96-100% for A-D, etc. (see the numbers in parentheses). This suggests that the top-k module can effectively find the true matches in D.

	Q	C	M_D	E	M_E	F	I
A-G	OL	8,388	291	4,063	190 (65.3)	166 (87.4)	40
	HASH	1,835	1,267	3,337	820 (64.7)	803 (97.9)	97
	SIM	7,406	192	4,341	104 (54.2)	73 (70.2)	29
	R	27,650	137	4,362	76 (55.5)	65 (85.5)	24
W-A	OL	210,782	87	6,570	48 (55.2)	37 (77.1)	7
	HASH	256,341	201	5,089	168 (83.6)	147 (87.5)	26
	SIM	46,900	135	7,089	56 (41.5)	46 (82.1)	7
	R	4,265	566	5,027	256 (45.2)	233 (91.0)	33
A-D	OL	56,869	41	4,270	41 (100.0)	37 (90.2)	8
	SIM	2,487	61	3,335	59 (96.7)	56 (94.9)	11
	R_1	3,764	41	3,843	41 (100.0)	38 (92.7)	10
	R_2	2,173	107	3,011	104 (97.2)	101 (97.1)	16
F-Z	OL	115	47	5,079	46 (97.9)	46 (100.0)	5
	HASH	10,165	52	4,653	51 (98.1)	51 (100.0)	5
	SIM	2,146	13	5,908	12 (92.3)	12 (100.0)	5
	R	124	33	5,239	32 (97.0)	32 (100.0)	5
M_1	OL	253,286	778	5,045	673 (86.5)	671 (99.7)	38
	HASH	212,296	188	4,948	100 (53.2)	100 (100.0)	13
	SIM	2,601,349	78	5,050	38 (48.7)	36 (94.7)	7
	R	89,344	202	5,213	113 (55.9)	109 (96.5)	11
M_2	$HASH_1$	11,115,136	4,530	5,428	661 (14.6)	648 (98.0)	47
	$HASH_2$	14,632,318	3,844	5,735	450 (11.7)	432 (96.0)	35
	SIM_1	27,461,378	2,220	5,420	1,012 (45.6)	1,012 (100.0)	54
	SIM_2	14,924,148	3,238	5,533	1,087 (33.6)	1,087 (100.0)	58
	SIM_3	8,512,446	4,228	5,587	1,151 (27.2)	1,151 (100.0)	61

Table 3: Accuracy in retrieving the killed-off matches.

Next we examine the Match Verifier. We want to know its accuracy if run until its natural stopping point (see Section 5). It is difficult to recruit enough real users for this large-scale experiment involving 25 blockers. So we use synthetic users, whom we assume can identify the true matches accurately (we describe multiple experiments with real users below).

Column F of Table 3 show that this module can retrieve a large number of matches in E, e.g., 65-803 for A-G (see the numbers outside parentheses), and that the retrieval rate is very high, e.g., 70-98% for A-G, 77-91% for W-A, etc. (see the numbers inside parentheses). Finally, Column I shows that the total number of iterations is 5-13 in 12 cases, 16-40 in 8 cases, 47-61 in 4 cases, and 97 in 1 case. The higher number of iterations is often due to the larger number of matches that have to be retrieved from E, e.g., for blocker HASH of dataset A-G, the module needed 97 iterations to retrieve 803 matches, a reasonable number of iterations given that each iteration shows only 20 tuple pairs to the user.

Thus, if the user runs the Match Verifier until its natural stopping point, he/she can retrieve a large number of matches. This is good news for applications in which blocker recall is critical, thus the user may want to examine *all* matches that the module can retrieve.

Accuracy & Explanations for the First Few Iterations: To examine if users can quickly find many matches and explanations, we asked volunteers to *manually work with the Match Verifier for the first three iterations.* Table 4 shows the results (for space reasons we only list five blockers for five datasets, the results for other blockers are similar). The table shows that the user needed only 7-10 mins to examine the first three iterations, was able to identify a large number of matches (28-43), and was able to identify multiple reasons for why they are killed off (a reason such as "large threshold (18)" means that tuple pair #18 was killed off due to the blocker using a large threshold, and this was the first pair where the user observed this problem). Overall, the results suggest that after examining the first few iterations, the

Blocker	# iteration	Label time	Blocker problems
OL (A-G)	3 iterations 31 matches	8 mins	large threshold (18); attribute "manuf" is sprinkled in the attribute "title" (18)
HASH (W-A)	3 iterations 43 matches	10 mins	different words for the same brand (6); missing values in attribute "brand" (13)
SIM (A-D)	3 iterations 28 matches	7 mins	large threshold (16); attribute "title" contains subtitle in one table (22)
R (F-Z)	3 iterations 32 matches	7 mins	different descriptions for attribute "type" (11); unnormalized attribute "address" (33); attribute "city" is sprinkled in "name" (47)
R (M_1)	3 iterations 41 matches	10 mins	input tables are not lower-cased (5); missing values in attribute "year" (12)

Table 4: Accuracy in the first 3 iterations and explanations.

user can already identify multiple problems with the blocker (which he/she can then fix).

6.2 Debugging State-of-the-Art Blockers

Suppose a user has manually developed a good standard blocker, or has used state-of-the-art techniques to learn a blocker, we want to know if MatchCatcher can still help improve the blocker's accuracy. Toward this goal, we performed two experiments.

Hash Blockers: First, we asked a user well-trained in EM to develop the best possible hash blockers for five datasets (the first five in Table 1). For example, for dataset A-G, this user created the blocker Q_1 which keeps a pair of tuples if they agree on "manufacturer" *or* on a hash of "price" *or* on a hash of "title". Thus, Q_1 combines three hash blockers. ([24] describes all five blockers in details.) We selected hash blocking because it is well-known, easy to understand, and fast. Hence it is considered a standard blocking method commonly used in practice. On the five datasets A-G, W-A, A-D, F-Z, and Music1, the best hash blockers achieve 75.6, 95.1, 100, 97.3, and 100% recall, respectively.

We then asked the same user to use MatchCatcher to try improving the above hash blockers. For A-D and Music1, which already have 100% recall, using MatchCatcher the user did not find any killed-off matches (as expected), so debugging terminated early. For A-G, W-A, and F-Z, however, debugging significantly improved recall from 75.6 to 99.7, 95.1 to 99.6, and 97.3 to 100%, respectively. [24] describes one such debugging scenario in details.

Learned Blockers: From a group of researchers we obtained Papers, the dataset described in the last row of Table 1. For this dataset, they have applied the method in [8] to learn blockers using a sample labeled by crowdsourcing, and we were able to obtain three such blockers (learned on three separate samples). The technical report [24] describes these blockers, which are the best blockers that the learning method has found in a very large space of blockers, including hash ones. Unfortunately, we do not have the entire set of "gold" matches for Papers (we do have some "gold" matches, but not all of them). Hence, we are unable to report recalls for these blockers.

We then asked a user to apply MatchCatcher to these blockers. After 5 iterations, the user found 76, 61, and 65 matches for the three blockers, respectively. More importantly, the user was able to identify a set of reasons for why these matches were killed off and suggestions for improving the blockers (see [24]). Given the lack of "gold" matches, we were not able to improve the blockers then compare their recalls. Nevertheless, the above experiments suggest that blockers learned using state-of-the-art solutions can still have many problems and MatchCatcher can help pinpoint these, to help the user improve recall.

6.3 MatchCatcher "in the Wild"

Over the past two years variations of MatchCatcher have been used by 300+ students in 4 data science classes and 7 EM teams at 6 organizations. The feedback has been overwhelmingly positive. For example, 18 teams used MatchCatcher in a class project, and reported that it helped (a) discovering data that should be cleaned, (b) finding the correct blocker types and attributes, (c) tuning blocker parameters, and (d) knowing when to stop. We have reported on some of this experience in [23]. Overall, we found that many real-world users have used MatchCatcher as *an integral part of an end-to-end blocker development process:* start with a simple blocker, use MatchCatcher to identify problems, improve the blocker, and so on, until MatchCatcher no longer reports substantial problems with the blocker.

6.4 Runtime & Scalability

MatchCatcher was implemented in Cython, and all experiments used a RedHat 7.2 Linux machine with Intel E5-1650 CPU. The top-k module took 6.6-9.4 secs (for dataset A-G), 97-310 (W-A), 2.8-3.2 (A-D), 0.2 (F-Z), 12.1-24.4 (M_1), 57-230 (M_2), and 65-344 (Papers), respectively. For the first five datasets, these times are quite small except 97-310 secs for W-A. On W-A, the k-th pair on the top-k list (recall that $k = 1000$) often has a very low score, e.g., 0.21-0.225. Thus the top-k module took more time. The last two datasets (M_2 and Papers) are much larger (500K tuples per table), and so took longer to run. In all cases, however, the total time is still under 5.8 minutes.

To examine how the top-k module scales, we measure its time as we vary the size of the two largest datasets, M_2 and Papers, at various percentages of the original datasets (which have 500-600K tuples per table). Figure 9 shows the results for the first three blockers in Table 3 for M_2 and all three blockers for Papers, for $k = 100$ (the left two plots) and $k = 1000$. The results show that the top-k module scales linearly or sublinearly as the table size grows. Finally, on all datasets the Match Verifier took under 0.1 sec to aggregate the top-k lists, and 0.14-0.18 secs to process user feedback in each iteration.

6.5 Additional Experiments

The technical report [24] describe extensive experiments on the performance of the MatchCatcher components, sensitivity analysis, and comparison with a recent related work. For space reasons we only briefly summarize those experiments here.

Performance of the Components: We show that using multiple configs instead of just one config significantly increases the number of retrieved matches, by 10-74%. Handling long attributes increases the recall of E (the fraction of matches in D that are in E) by up to 11%, compared to not handling them in config generation. Our experiments also show that the joint top-k processing strategy over multiple configs significantly outperforms the baseline of executing each config individually, by as much as 3.5 times. Finally, we found that active/online learning significantly outperforms weighted median ranking in the Match Verifier.

Sensitivity Analysis: We found that increasing k (the number of pairs retrieved per config) does increase the number of true matches retrieved, but only up to a certain k, and comes at the cost of higher runtime, and that using 3 active learning iterations (as we currently do) provides a good balance between increasing the classifier accuracy and increasing recall in the Match Verifier.

Comparison with Recent Work: We found MatchCatcher significantly outperforms the work in [29], which uses a single config, e.g., improving the recall of E by 26-47% on the A-G dataset.

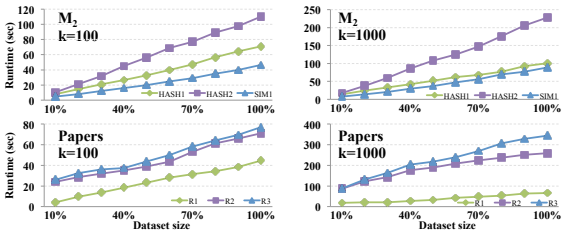

Figure 9: Runtime of top-k module for varying table sizes.

7 ADDITIONAL RELATED WORK

We have discussed related work throughout the paper. We now discuss additional related work. As far as we can tell, our recent work [23] is the first to raise the need for debugging for blocking. But that work focuses on developing end-to-end EM systems. It does not discuss any debugging solution in depth, as we do here. Other related works include debugging for data cleaning [17], schema mapping [5], and data errors in spreadsheets [1]. They do not address EM and their solutions do not apply to our context. But they do underscore the importance of debugging for data integration and cleaning.

SSJs have received much attention, e.g., [21, 35] (see [36] for a survey). Top-k SSJs are studied in [34, 37]. [37] proposes a B+ tree based index to scale top-K SSJs on edit distance. It does not work well for datasets with large textual difference [36], however, a common occurrence in our case. The work [34], which uses prefix filtering to find the top-k pairs, is better suited to our case. But it does not handle long strings well [36]. Here we have significantly improved this work and extended it to work over multiple configs.

The idea of computing the similarity score of a string pair only if their prefixes share at least q tokens (see Algorithm QJoin in Section 4.1) is also discussed in [32]. That work however focuses on SSJs with thresholding, e.g., matching two strings x and y if $jaccard(3g(x), 3g(y)) \geq \alpha$. Its solution uses threshold α to find the optimal q, and is not applicable to the top-k context considered in this paper (which has no threshold α).

The work [27] describes a blocking method that performs a variation of weighted overlap blocking to find tuples that are highly similar string-wise. MatchCatcher however does not use this method in top-k SSJs because it is not clear how to modify it to enable reuse (among the different configs). The work [16] is related to our work on config generation in that it defines the notion of matching dependencies, using which we can deduce a set of attributes for comparing tuple pairs. However, it is not applicable to our context because it requires the user to manually specify matching dependencies, using domain knowledge, in a potentially time consuming process.

Rank aggregation has been studied extensively in the database/IR communities, e.g., [4, 11, 15]. Active learning (AL) for EM has been studied in [18, 26, 28]. But they perform extensive AL to learn an accurate matcher. In contrast, we use only a few AL iterations to learn a classifier with reasonable accuracy, then use it to surface matches for debugging purposes. The above work also does not combine AL with online learning as we do. Finally, the work [31] uses a learning-based UI model similar to ours, but for IR tasks.

8 CONCLUSIONS & FUTURE WORK

We have shown that debugging blocker accuracy is critical for EM, and have described MatchCatcher, a solution to this problem. As for future work, in certain cases the user may find a large number of killed-off matches. So we plan to develop a method to automatically explain why each match is killed off by the blocker, summarize these explanations, then present the summary to the user. When fixing a problem affecting a killed-off match, the user may want to know how pervasive this problem is (and focus on fixing the most pervasive ones first). For this purpose, given a killed-off match, we plan to develop a method to find all tuple pairs that are similar to that match (from a blocking point of view).

REFERENCES

[1] D. Barowy, D. Gochev, and E. Berger. 2014. CheckCell: data debugging for spreadsheets. OOPSLA.
[2] M. Bilenko, B. Kamath, and R. J. Mooney. 2006. Adaptive blocking: learning to scale up record linkage. ICDE.
[3] M. Bilenko and R. J. Mooney. 2003. Adaptive duplicate detection using learnable string similarity measures. SIGKDD.
[4] B. Brancotte, B. Yang, G. Blin, S. Cohen-Boulakia, A. Denise, and S. Hamel. 2015. Rank aggregation with ties: experiments and analysis. VLDB.
[5] L. Chiticariu et al. 2006. Debugging schema mappings with routes. VLDB.
[6] P. Christen. 2012. *Data Matching.* Springer.
[7] P. Christen. 2012. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE TKDE* 24, 9 (2012), 1537–1555.
[8] S. Das et al. 2017. Falcon: scaling up hands-off crowdsourced entity matching to build cloud services. SIGMOD.
[9] A. Doan, A. Halevy, and Z. Ives. 2012. *Principles of Data Integration.* Elsevier.
[10] X. Dong, A. Halevy, and J. Madhavan. 2005. Reference reconciliation in complex information spaces. SIGMOD.
[11] C. Dwork et al. 2001. Rank aggregation methods for the web. WWW.
[12] V. Efthymiou, G. Papadakis, et al. 2017. Parallel meta-blocking for scaling entity resolution over big heterogeneous data. *Inf. Syst.* 65 (2017), 137–157.
[13] V. Efthymiou, K. Stefanidis, and V. Christophides. 2016. Benchmarking blocking algorithms for web entities. *IEEE Transactions on Big Data* (2016).
[14] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. 2007. Duplicate record detection: a survey. *IEEE Trans. Knowl. Data Eng.* 19, 1 (2007), 1–16.
[15] R. Fagin, R. Kumar, and D. Sivakumar. 2003. Efficient similarity search and classification via rank aggregation. SIGMOD.
[16] W. Fan et al. 2009. Reasoning about record matching rules. VLDB.
[17] H. Galhardas, D. Florescu, D. Shasha, et al. 2000. AJAX: an extensible data cleaning tool. SIGMOD.
[18] C. Gokhale, S. Das, A. Doan, J. F. Naughton, N. Rampalli, J. Shavlik, and X. Zhu. 2014. Corleone: hands-off crowdsourcing for entity matching. SIGMOD.
[19] Y. Govind et al. 2017. CloudMatcher: A Cloud/Crowd Service for Entity Matching. BIGDAS@KDD.
[20] M. A. Hernández and S. J. Stolfo. 1998. Real-world data is dirty: data cleansing and the merge/purge problem. *Data Min. Knowl. Discov.* 2, 1 (1998), 9–37.
[21] Y. Jiang, G. Li, J. Feng, et al. 2014. String similarity joins: an experimental evaluation. VLDB.
[22] L. Kolb, A. Thor, and E. Rahm. 2011. Parallel sorted neighborhood blocking with MapReduce. BTW.
[23] P. Konda et al. 2016. Magellan: toward building entity matching management systems. VLDB.
[24] H. Li et al. 2017. *MatchCatcher: a debugger for blocking in entity matching.* Technical Report. http://pages.cs.wisc.edu/~anhai/papers1/matchcatcher-tr.pdf.
[25] M. Michelson. 2006. Learning blocking schemes for record linkage. AAAI.
[26] B. Mozafari, P. Sarkar, M. Franklin, M. Jordan, and S. Madden. 2014. Scaling up crowd-sourcing to very large datasets: a case for active learning. VLDB.
[27] G. Papadakis, G. Koutrika, T. Palpanas, and W. Nejdl. 2014. Meta-blocking: taking entity resolutionto the next level. *IEEE TKDE* 26, 8 (2014), 1946–1960.
[28] S. Sarawagi and A. Bhamidipaty. 2002. Interactive deduplication using active learning. SIGKDD.
[29] D. Song and J. Heflin. 2011. Automatically generating data linkages using a domain-independent candidate selection approach. ISWC.
[30] K. Stefanidis, V. Efthymiou, M. Herschel, and V. Christophides. 2014. Entity resolution in the web of data. WWW (Companion Volume).
[31] A. Tian and M. Lease. 2011. Active learning to maximize accuracy vs. effort in interactive information retrieval. SIGIR.
[32] J. Wang, G. Li, and J. Feng. 2012. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. SIGMOD.
[33] S. E. Whang, D. Menestrina, G. Koutrika, M. Theobald, and H. Garcia-Molina. 2009. Entity resolution with iterative blocking. SIGMOD.
[34] C. Xiao, W. Wang, X. Lin, and H. Shang. 2009. Top-k set similarity joins. ICDE.
[35] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. 2011. Efficient similarity joins for near-duplicate detection. *TODS* 36, 3 (2011), 15.
[36] M. Yu, G. Li, D. Deng, and J. Feng. 2016. String similarity search and join: a survey. *Frontiers of Computer Science* 10, 3 (2016), 399–417.
[37] Z. Zhang, M. Hadjieleftheriou, B. Ooi, et al. 2010. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. SIGMOD.