Sparkly: A Simple yet Surprisingly Strong TF/IDF Blocker for Entity Matching

(Technical Report)

Derek Paulsen[1,2], Yash Govind[2], AnHai Doan[1,2]
[1]University of Wisconsin-Madison, [2]Informatica

ABSTRACT

Blocking is a major task in entity matching (EM). Numerous blocking solutions have been developed, but as far as we can tell, blocking using the well-known tf/idf measure has received virtually no attention. Yet, when we experimented with tf/idf blocking using Lucene, we found it does quite well. So in this paper we examine tf/idf blocking in depth. We develop Sparkly, which uses Lucene to perform top-k tf/idf blocking in a distributed share-nothing fashion on a Spark cluster. We develop techniques to identify good attributes and tokenizers that can be used to block on, making Sparkly completely automatic. We perform extensive experiments showing that Sparkly outperforms 8 state-of-the-art blockers. Finally, we provide an in-depth analysis of Sparkly's performance, regarding both recall/output size and runtime. Our findings suggest that (a) tf/idf blocking needs more attention, (b) Sparkly forms a strong baseline that future blocking work should compare against, and (c) future blocking work should seriously consider top-k blocking, which helps improve recall, and a distributed share-nothing architecture, which helps improve scalability, predictability, and extensibility.

PVLDB Reference Format:
Derek Paulsen[1,2], Yash Govind[2], AnHai Doan[1,2]. Sparkly: A Simple yet Surprisingly Strong TF/IDF Blocker for Entity Matching (Technical Report). PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at URL_TO_YOUR_ARTIFACTS.

1 INTRODUCTION

Entity matching (EM) finds data instances that refer to the same real-world entity, such as (David Smith, Univ of Wisc) and (D. Smith, UW-Madison). Most EM solutions proceed in two steps: blocking and matching. Given two tables A and B to match, the *blocking step* uses heuristics to quickly remove tuple pairs ($a \in A, b \in B$) judged unlikely to match. The *matching step* then applies a matcher to the remaining tuple pairs to predict match/no-match.

Both the blocking and matching steps have received significant attention (e.g., [2, 6–8, 13, 15, 28, 30, 31, 33]). In this paper we focus on the blocking step. Over the past 30 years, numerous blocking solutions have been developed. The goal is to maximize *recall* (the fraction of true matches in the blocking output) while minimizing *the output size* and *the runtime*. Earlier blocking solutions include sorted neighborhood, attribute equivalence, and hash-based solutions. Later solutions use similarity measures such as edit distance, Jaccard, and cosine. Recent work has developed more sophisticated solutions, such as meta blocking, rule-based, and deep learning based blocking (see Section 2).

In the past few years, as a part of the Magellan project at UW-Madison, which develops an open-source EM platform [20], we have implemented many blocker types, develop new blocker types [40], and applied them to real-world EM tasks in domain sciences and industry [17]. While doing this, we found that a relatively simple blocking solution that uses the tf/idf similarity measure, as implemented in the open-source Apache Lucene library, seems to work quite well.

This is rather surprising because as far as we can tell, tf/idf based blocking has received very little attention. For example, the book "Data Matching" [7] and several recent EM surveys [6, 15, 30, 33] briefly discuss only a single tf/idf solution proposed 20 years ago [26]. This solution runs on a single machine, is difficult to scale (see Section 6), and is shown experimentally to perform worse than other solutions [6]. Since then we are not aware of any work that examines tf/idf blocking. Yet, not only that we found tf/idf blocking promising, we have also heard anecdotes of its being used at several companies.

As a result, in this paper we perform an in-depth examination of tf/idf blocking. We begin by developing a solution called Sparkly Manual, which takes as input two tables A and B with the same schema, and outputs tuple pairs ($a \in A, b \in B$) judged likely to match. There are two key ideas underlying Sparkly Manual. First, *it performs top-k blocking*. For each tuple t of the larger table, say B, it finds the top k tuples in A with the highest tf/idf scores (where k is pre-specified), then pairs these tuples with t and outputs the pairs.

The second idea underlying Sparkly Manual is that *it performs the above top-k computations in a distributed shared-nothing fashion*, using Lucene on a Spark cluster (hence the name Sparkly, which stands for Spark + Lucene + Python). Specifically, it uses Lucene to build an inverted index I for table A on the driver node of the Spark cluster, ships the index I to all worker nodes, distributes the tuples of table B to the worker nodes, then uses Lucene to perform top-k computations for the tuples at the worker nodes. Thus, *the worker nodes operate in parallel and share no dependencies.* Each node processes a subset of tuples in B.

We compare Sparkly Manual with 8 state-of-the-art (SOTA) blockers on 15 datasets that have been extensively used in recent EM work [27, 36, 40]. The 8 blockers are: 2 deep learning (DL)

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

blockers [40], 1 blocker combining a DL blocker and a rule-based blocker [40], 3 token-based blockers from the well-known JedAI open-source EM platform [32, 34], 1 kNN blocker identified as highly promising by a recent work [36], and a variation of this kNN blocker. *Surprisingly, Sparkly Manual outperforms all of the above blockers.* It achieves higher or comparable recall at a much smaller output size, and the performance gap is quite significant in several cases (see the experiment section).

While appealing, Sparkly Manual has a limitation. It requires the user to manually identify the attributes to be blocked on, e.g., product title, or name and phone (hence the word "Manual" in its name). Then it computes the tf/idf score between any two tuples $a \in A, b \in B$ using only these attributes, after 3-gram tokenization.

It can be difficult for users to identify good blocking attributes. So we develop Sparkly Auto, which automatically identifies a set of good blocking attributes, together with an appropriate tokenizer for each attribute (e.g., 3-gram, word-level). Our key observation is that *a good blocking attribute helps to discriminate between matches and non-matches.* We propose techniques to quantify discriminativeness, then to effectively search a large space for the optimal combination of attributes and tokenizers that maximizes this quantity.

We show that *Sparkly Auto achieves comparable or higher recall than Sparkly Manual, yet runs much faster* (Section 4 explains why). In particular, Sparkly Auto can block large datasets at reasonable time and cost, e.g., blocking tables of 10M tuples under 100 minutes on an AWS cluster of 10 commodity nodes, costing only $12.5. This suggests that Sparkly Auto can already be practical for many real-world EM problems.

We conclude by discussing questions that arise in light of Sparkly's strong performance. First, we speculate why tf/idf blocking has received little attention so far. Second, we analyze factors leading to high recall for Sparkly, demonstrating that this is due to top-k blocking and use of tf/idf measure. We also analyze possible reasons for why deep learning blockers underperform Sparkly. Third, we analyze factors leading to fast runtime for Sparkly. We argue that this is due to the distributed share-nothing architecture, which allows Sparkly to scale out, and the recently developed block-max WAND technique [4, 11, 12], which allows Lucene to perform top-k search very fast [19]. Fourth, we discuss cases where Sparkly may fail to achieve high recall. Finally, we identify promising future research directions. In summary, the contributions and takeaways of this paper are as follows:

- We develop Sparkly, a tf/idf blocker that uses Lucene to perform top-k blocking in a distributed share-nothing fashion on a Spark cluster. We develop techniques to identify good attributes and tokenizers to block on.
- We perform extensive experiments showing that Sparkly outperforms 8 state-of-the-art blockers. This is rather surprising because tf/idf blocking has received very little attention. *The takeaway is that tf/idf blocking needs more attention, and that Sparkly forms a strong baseline that future blocking work should compare against.*
- We provide an in-depth analysis of Sparkly's performance, regarding both recall/output size and runtime. *The takeaway is that future blocking work should consider top-k blocking, which helps improve recall, and a distributed share-nothing architecture, which helps improve scalability, predictability, and extensibility.*
- Based on the above analysis, we identify promising research directions for blocking.

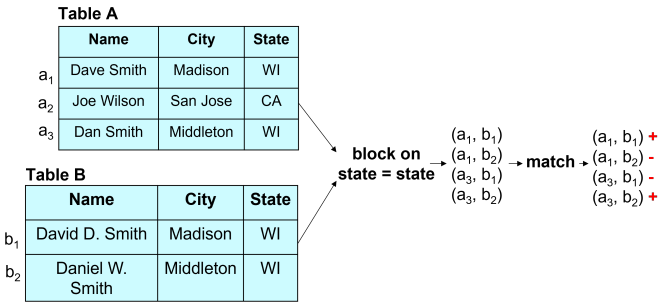

Figure 1: The blocking and matching steps of EM.

The page [35] provides the Sparkly code and all experiment datasets (except Hospital, which is private).

2 BLOCKING FOR ENTITY MATCHING

In this section we introduce the problem of blocking for EM, discuss existing solutions, and ways to evaluate blocking solutions.

EM, Blocking, Matching: Many EM variations exist [7, 13]. A common EM variation [40], which we consider in this paper, is as follows: given two tables A and B with the same schema, find all tuple pairs ($a \in A, b \in B$) that refer to the same real-world entity. We call these pairs *matches*.

Considering all pairs in $A \times B$ takes too long for large tables. So EM is typically performed in two steps: *blocking* and *matching* [7, 13]. *The blocking step* uses heuristics to quickly remove a large number of pairs judged unlikely to match. *The matching step* applies a rule- or ML-based matcher to each remaining pair, to predict match or non-match. Figure 1 illustrates these steps. Here blocking keeps only those pairs that share the same state. In this paper we focus on the blocking step.

Existing Blocker Types, Threshold vs. Top-k Blocking: Over the past 50 years numerous blocking solutions have been developed (see [6, 15, 30, 33] for surveys). They fall roughly into five types: *sort, hash, similarity-based, rule-based*, and *composite. Sorted neighborhood* computes for each tuple a key, sorts tuples based on keys, then outputs a pair of tuples if their keys are within a pre-defined distance.

Hash-based methods compute for each tuple one or multiple hash values (a.k.a. keys), groups all tuples sharing the same hash value into a *block*, then outputs a pair of tuples if they belong to the same block. Examples of such methods include attribute equivalence, phonetic blocking, suffix array, etc. [6, 30, 33]. Most existing blocking methods are hash-based.

Similarity-based methods output only those tuple pairs where the similarity score between the tuples exceeds a pre-specified threshold, or one tuple is within the kNN (k-nearest) neighborhood of the other tuple [36]. We refer to these options as *threshold blocking* and *top-k blocking*, respectively. Sparkly uses top-k blocking, which we show experimentally to be critical to achieve high recall.

Similarity scores that have been considered for blocking include syntactic scores such as Jaccard, cosine, edit distance [13], and semantic scores such as those computed using word embedding/deep learning (DL) techniques [40].

Rule-based methods employ multiple blocking rules, where each rule can employ multiple predicates (e.g., if the Jaccard score of the titles is below 0.6 and the years are not equivalent, then the two papers do not match) [16]. Given a set of rules, the blocker figures out the best way to create a workflow and execute it using indexes [16]. Finally, *composite methods* generalizes rule-based blocking and can combine multiple blocking methods in a complex *pre-specified* pipeline. Examples include canopy blocking [13] and the union of a DL method with a rule-based method in [40].

Recent Research Directions: In recent years researchers have pursued several directions regarding the above five blocker types [30, 33]. They have examined how to scale blocking methods (e.g., using Hadoop/Spark) [10] and how to apply DL (e.g., to develop novel hash-based [14] and similarity-based blockers [40]).

As discussed earlier, hash-based methods generate *blocks of tuples*, then output pairs whose tuples belong to the same block. A novel recent direction, called *meta-blocking*, examines how to manage these blocks (e.g., remove/prune blocks) [33]. A related direction is *token blocking*, in which each block contains all tuples that share a particular token. These blocks can be managed using meta blocking. Another interesting direction, called *schema-agnostic*, does not assume that tables A and B share the same schema. The well-known JedAI EM platform implements many meta-blocking, token-blocking, and schema-agnostic techniques [32, 34]. Other important directions include learning blockers (e.g., rule-based ones using tuple pairs labeled match/no-match [16]), using the feedback from the matcher to improve the blocker, and explaining blockers [6, 30, 33].

Evaluating Blockers: Most existing works evaluate blockers in three aspects: *recall, output size*, and *runtime*. Let $G \subseteq A \times B$ be the set of (unknown) gold matches, and $C \subseteq A \times B$ be the set of tuple pairs output by a blocker Q. Then the *recall* of Q is $|C \cap G|/|G|$, the fraction of gold matches in the output of Q. The *output size* is $|C|$, and the *runtime* is measured from when the blocker receives the two tables A and B until when it outputs C.

Other aspects considered important, especially in industry, include the ease of tuning, the ability to block on arbitrarily large tables (e.g., those with billions of tuples) without crashing, extensibility (e.g., with more blocking methods/rules), the ability to estimate the total blocking time, explainability, and the ability to run the blocker easily in a variety of environments (e.g., a single laptop, a Spark cluster, a Kubernetes cluster), among others.

In this paper, we will evaluate blockers using the above three popular aspects: recall, output size, and runtime. We will briefly discuss Sparkly regarding additional aspects in Section 5, but deferring a thorough evaluation of these aspects to future work.

3 THE SPARKLY SOLUTION

In this section we describe the tf/idf measure used in keyword search (KWS), the open-source KWS library Lucene, then Sparkly, which uses Lucene to perform blocking for EM.

3.1 The TF/IDF Family of Scoring Functions

TF/IDF is a well-known family of scoring functions for ranking documents in KWS [25]. To explain, consider a set of documents $\mathcal{D} = \{D_1, \ldots, D_N\}$, where each document D_i is a string (e.g., article, email). Given a user query Q, which is also a string, we want to find documents in $\mathcal{D}$ that are most relevant to Q. To do so, we compute a score $s(D, Q)$ for each document D, then return the documents ranked in decreasing score.

A well-known scoring function [13] is as follows. First we tokenize each document D into a bag of *tokens*, also called *terms*. For example, "cat chases mouse" can be tokenized into a bag of word-level tokens {"cat", "chases", "mouse"}, or a bag of 3-grams {"##c", "#ca", "cat", ..., "e##"}.

Next, we convert document D into a vector V_D of *weights*, one weight per term, where the weight for term t is $V_D(t) = tf(t, D) \cdot idf(t)$. Here $tf(t, D)$ is *the frequency of term t* in document D, i.e., the number of times it occurs in D. The quantity $idf(t)$ is *the inverse document frequency of term t*, defined as $log(N/df(t))$, where N is the number of documents in $\mathcal{D}$, and $df(t)$ is the number of documents that contain term t.

We tokenize and convert query Q into a vector of weights V_Q in a similar fashion. Finally, we compute score $s(D, Q)$ to be the cosine of the angle between the two vectors V_D and V_Q:

$$s(D, Q) = [\sum_t V_D(t) \cdot V_Q(t)]/[\sqrt{\sum_t V_D(t)^2} \cdot \sqrt{\sum_t V_Q(t)^2}], \quad (1)$$

where t ranges over all terms in D and Q.

The above tf/idf definition captures the intuition that if a term t of query Q occurs often in a document D, then D is likely to be relevant to Q and score $s(D, Q)$ should be high. This is reflected in the use of the term frequency $tf(t, D)$. A higher $tf(t, D)$ leads to a higher weight for t in V_D, and consequently a higher $s(D, Q)$. But this should not be true if term t also occurs in many other documents (e.g., common words such as "and", "the", "inc", "str"). In such cases term t should be discounted, i.e., its weight in V_D should be low, and this is accomplished by multiplying the term frequency $tf(t, D)$ with the inverse document frequency $idf(t)$.

Over the years, many tf/idf scoring functions have been proposed. Among them, the following function, called *Okapi BM25*, has become most popular, and is the default scoring function used by Lucene [38]:

$$s(D, Q) = \sum_{t \in Q} \frac{tf(t, D) \cdot (k_1 + 1)}{tf(t, D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{avgdl})} \cdot idf(t), \quad (2)$$

where $idf(t) = log(\frac{N - df(t) + 0.5}{df(t) + 0.5} + 1)$, and k_1 and b are free parameters, often set as $k_1 \in [1.2, 2.0]$ and $b = 0.75$.

In the above scoring function, each term t in query Q has a score, and the BM25 score is the sum over the scores of all terms in Q. The score of each term t is designed to capture the following intuition. First, once a document D already has a high number of occurrences of term t, more occurrences should not significantly raise the score of t. This is accomplished by dividing $tf(t, D)$ by $tf(t, D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{avgdl})$ (see Equation 2). Here parameter k_1 controls how much additional occurrences of term t should raise its score.

Second, for the same number of occurrences of term t in document D, longer D should be penalized, because it is less likely to be about t. For example, if D mentions "cats" once and is really short, then it is likely to be about cats, but if D is really long, then it is unlikely to be about cats. This is accomplished by the quantity $(1 - b + b \cdot \frac{|D|}{avgdl})$ in Equation 2. Here parameter b controls how much the longer document should be penalized. The article [38] provides a detailed explanation of the intuition behind BM25, which has been shown to work quite well for KWS [19, 25].

3.2 The Lucene KWS Library

Over the years, many open-source software for KWS have been developed. Among them Apache Lucene, first released in 1999, has become most popular [19]. The latest releases of Lucene, since 2015, have used state-of-the-art techniques in KWS to be both accurate and fast [19].

Specifically, Lucene uses BM25 as the default scoring function, ensuring highly accurate KWS results. It has also been extensively optimized, to be very fast for top-k querying, i.e., *given a query Q and a set of documents $\mathcal{D}$, find the top k documents in $\mathcal{D}$ that have the highest BM25 score with Q,* for a pre-specified k (typically up to a few hundreds). To do this, naively we can use an inverted index to find all documents in $\mathcal{D}$ that share at least one term with Q, compute BM25 scores for all of them, then sort and return the top k documents. This however would be very slow, because the set of documents sharing at least one term with Q is often very large.

To solve this problem, Lucene uses a recently developed KWS technique called *block-max WAND* [4, 11, 12]. Briefly, an inverted index consists of multiple *postings*, one per term. The posting $t : (id_1, f_1), \ldots, (id_n, f_n)$ lists the IDs of all documents in $\mathcal{D}$ that contain term t, together with the frequency of t in each document (in practice postings often contain additional information). During the indexing process, when creating this posting, Lucene partitions the list of IDs into *blocks*, and assigns to each block a *max number*, which is an upper bound on the score that term t can contribute to the BM25 score of any document in the block. Thus, a block with two documents id_1 and id_2 and a max number 3 means that term t contributes at most an amount of 3 to the BM25 score of id_1 and id_2.

At query time, Lucene performs a *branch-and-bound search* to find the top k. It maintains a list of top-k documents found so far. It uses the max number of a block B to derive an upper bound on the BM25 score of *any document* in B. If this upper bound shows that none of the documents in B can make it into the top k, then Lucene will skip computing the BM25 scores for all documents in B. This way, Lucene avoids examining a huge number of documents, and can generally find the top-k documents very fast, as we will see in the experiment section.

Lucene has become the library of choice for a wide variety of KWS applications. Two other popular open-source KWS systems, Solr and ElasticSearch, build on Lucene. *As a library, Lucene provides two key API functions: indexing and querying.* Solr (started in 2004) and ElasticSearch (started in 2010) use these API functions, but provide extensive support for indexing and querying a large number of documents on a cluster of machines.

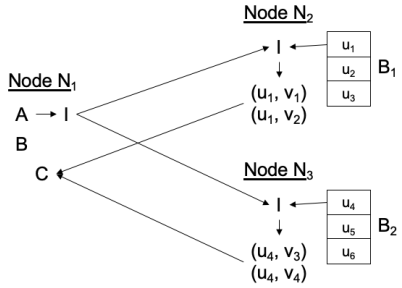

Figure 2: Sparkly's execution on a 3-node cluster.

3.3 The Sparkly Solution

We now describe Sparkly, which takes as input two tables A and B with the same schema, and outputs a table C consisting of tuple pairs $(a \in A, b \in B)$ judged likely to match. (The Appendix discusses how Sparkly can handle two tables with differing schemas or deduplication.)

To do so, Sparkly uses two key ideas. First, *it performs top-k blocking*. Specifically, it builds an inverted index I for the smaller table, say table A. Then for each tuple b in table B, it probes I to find the top k tuples in A with the highest tf/idf scores (where k is pre-specified), then pairs these tuples with b and outputs the pairs.

Second, Sparkly *executes the above steps in a distributed share-nothing fashion, using Lucene on a Spark cluster*. We now describe the execution in detail, using the 3-node Spark cluster in Figure 2. (Later we discuss the rationales behind the design decisions.)

Build the inverted index I of table A: Suppose that tables A and B reside on the primary node N_1 (see Figure 2), and that A is the smaller table, i.e., having fewer tuples than B. Sparkly chops table A horizontally into multiple chunks, each containing multiple tuples, starts multiple threads on the entire Spark cluster, sends each chunk to a thread, which calls Lucene's indexing procedure to create an inverted index for that chunk. Sparkly then combines these inverted indexes into a single inverted index I for table A, and writes I to the local disk of node N_1.

Ship index I and tuples of table B to the secondary nodes: Sparkly then ships index I to the local disks of the secondary nodes N_2 and N_3 (see Figure 2). Next, it chops table B (on primary node N_1) into chunks, each containing multiple tuples (currently set to 500), send each chunk to a secondary node and assign to a thread on that node. Figure 2 shows that a chunk B_1 of table B consisting of tuples u_1, u_2, u_3 is sent to a thread on node N_2, and that another chunk B_2 consisting of tuples u_4, u_5, u_6 is sent to a thread on node N_3.

Find top-k tuples in table A for each tuple of table B: Each thread now goes through the tuples in the assigned chunk. For each tuple, it probes index I to find the top k tuples in table A with the highest tf/idf scores, pair these tuples with the probing tuple, then sends the pairs back to the primary node N_1. (The thread only sends back the IDs, not the full tuples.)

Consider again the thread for chunk B_1 with tuples u_1, u_2, u_3 (under "Node N_2" in Figure 2). Suppose $k = 2$. This thread first processes tuple u_1: it probes index I to find the top 2 tuples in table A with the highest tf/idf scores with u_1. Suppose these are tuples v_1, v_2. Then the thread creates the pairs $(u_1, v_1), (u_1, v_2)$ and send

them back to node N_1 (see the figure). Next, the thread processes tuple u_2, then tuple u_3. Similarly, Figure 2 shows how a thread on node N_3 processes chunk $B_2 = \{u_4, u_5, u_6\}$.

When Sparkly has processed all chunks of table B, and all pairs sent back from the secondary nodes have been collected into a table C on primary node N_1, Sparkly terminates, returning C as the blocking output.

The tf/idf scoring function: All that is left is to describe the scoring function used by Sparkly. First, we ask that the user *manually* identify a set of attributes to block on (later we discuss how to *automatically* identify this set). Typically these are "identity" attributes, such as name, phone, address, product title, brand, etc.

Next, for each tuple (in table A or B), we concatenate the values of these attributes into a single string s, lowercase all characters in s, tokenize s into a bag of 3-gram tokens, and remove all non-alphanumeric tokens. For example, if the string s for a tuple is "David Smith 457-6983", then we convert it into the bag of 3-grams {##d, #da, dav, ..., 457, ..., 83#, 3##}, then remove non-alphanumeric 3-grams such as "57-", "7-6".

Let B_t be the bag of 3-grams for a tuple t. When indexing table A, for each tuple $t \in A$, we index only B_t, not the entire tuple t. Finally, when querying, we compute the BM25 score between two tuples u, v to be the BM25 score between B_u and B_v.

Discussion: We now discuss the rationales behind the design decisions of Sparkly.

Use top-k instead of thresholding: This is *the most important decision that we made.* As discussed earlier, existing similarity-based blocking works output a tuple pair using one of the two conditions: (1) the similarity score between the two tuples exceeds a pre-specified threshold α, or (2) one tuple is among the top k tuples with the highest similarity scores with the other tuple, for a pre-specified k.

We use top-k instead of thresholding for the following reasons. First, it is often easier to select a value for k than a value for threshold α. Given a value for k, we know precisely how big the blocking output will be, and the rule of thumb is to select k that produces the largest blocking output that the matching step can handle, because the larger the blocking output, the higher the recall. On the other hand, we often do not have any guidances on how to select a good threshold α.

Second, we observe that real-world data is often so noisy that the similarity scores of many matching tuple pairs can be quite low (see Section 5). This makes it difficult to set threshold α. A high threshold kills off many matches, producing low recall. A low threshold often blows up the blocking output size in unpredictable ways. In contrast, in such cases we observe that the matching tuples are often still within the top-k "distance" of each other, making top-k retrieval still effective, as we show in Section 4.

Finally, Lucene and many other KWS systems are highly optimized runtime-wise for top-k search, but not for threshold search.

Do top-k on just one side instead of both sides: Currently we do top-k only from table B into table A, i.e., for each tuple in B find the k tuples in A with the highest tf/idf scores. Another option is to do top-k on both sides: from B into A and from A into B, then return the union of the two outputs. We experimented with this option but found that it can significantly increase runtime yet improve recall only minimally. It also complicates coding (e.g., we have to write code to remove duplicate pairs from the outputs of both sides).

Do top-k from the larger table: We index the smaller table, say table A, then do top-k probing from the larger table B because indexing the smaller table takes less time and produces a smaller inverted index I. Shipping this smaller index I to the secondary Spark nodes takes less time. Finally, probing from the larger table rather than the smaller one tends to produce higher recall, given the same k value.

Ship the index and tuples of table B to the secondary nodes: This is *the second most important decision that we made.* The challenge here is to find an efficient way to do distributed top-k probing on a Spark cluster. Toward this goal, recall that we create the inverted index I for table A on the primary node N_1. Table B also resides on N_1. So the simplest solution is to do all top-k probings there, using only the cores of N_1. However, N_1 has a limited number of cores (e.g., 16, 32), so it can run only a limited number of threads, severely limiting how much top-k probing we can do in parallel.

The next solution is to send the tuples of B to the secondary Spark nodes, then do top-k probing from the secondary nodes into the index I on primary node N_1. This way, the secondary nodes can run a much larger number of threads. Unfortunately, when these threads contact primary node N_1 to do top-k probing, they would need to rely on the threads running on the cores of N_1 to do the actual probing into index I. So once again, the limited number of threads on N_1 becomes the bottleneck for scaling.

As a result, we decided to ship the index I and the tuples of B to the secondary nodes. Each secondary node then runs multiple threads, each doing top-k probing using the copy of I on that node. So we can do as many top-k probings in parallel as the number of threads on the secondary nodes. This produces a distributed share-nothing solution that is highly modular and can scale horizontally as we add more secondary Spark nodes. In particular, given n worker nodes, m threads on each node (worker or driver), and p tuples in each chunk processed by a thread, the runtime of Sparkly can be estimated as

$$t(Sparkly) = \frac{|A|}{m} t_{index} + t_{shipI} + \frac{|B|}{pnm}(t_{query} + t_{shipR}), \quad (3)$$

where t_{index} is the average time to index a tuple in A, t_{query} is the average time to perform top-k querying for all tuples in a chunk, and t_{shipR} is the average time to ship the results of top-k querying (for a chunk) back to the driver node.

Partitioning very large tables A and B: A major concern is whether shipping index I would take too long, because it can be very large. This turned out not to be the case. For example, in our experiments, indexing a table of 10M tuples produces indexes of size 1.3-2GB, and shipping these takes 21-32 seconds (see Section 4).

Still, one may ask what if the tables have 500M or 5B tuples? Would the indexes become too big to fit on the disks of Spark nodes? *Our solution is to break table A (the smaller table, to be indexed) into partitions of say 50M tuples, then process the partitions sequentially.* For example, if table A has 100M tuples, then we break A into partitions A_1 and A_2 each having 50M tuples, then run two blocking tasks: A_1 vs. B and A_2 vs. B. Finally, we combine the top-k

results produced by these tasks. *This guarantees that Sparkly never has to build and ship indexes for more than 50M tuples.* We have used this solution to successfully block tables of hundreds of millions of tuples (details omitted for confidentiality reasons). In practice, most EM needs that we have seen involve tables of fewer than 50M tuples and does not even require partitioning.

Use Lucene instead of ElasticSearch or Solr: We use Lucene because it provides highly effective procedures to index a table and do top-k probing, which are exactly what we need. ElasticSearch (ES) and Solr build on top of Lucene and provide a lot more capabilities that we do not need (e.g., sharding) yet can cause complications.

For example, when we first built Sparkly, we used ES. The simplest way to use ES is to run an ES cluster, i.e., ES will control all nodes in the cluster and decide where to place data and perform processing. However, it is difficult to extend this solution, e.g., by adding pre-processing and post-processing steps. As a result, we decided to run a Spark cluster, in which ES is installed in each worker node. This makes it much easier to extend the solution.

But then we found that it was difficult to run this solution in a Kubernetes cluster (a common setting in the industry). This is because Kubernetes controls how and where data is sent, and when we perform a top-k query, we have no way to guarantee that this query will go to the ES instance installed on the same node. It is possible that this query will be sent via the network to an ES instance installed on some other node. In other words, we cannot guarantee *data locality*, and this can seriously slow down top-k querying. In addition, it takes much longer (and more pain) to install Sparkly, because we had to install ES as a part of the process. So we switched to Lucene, which addresses the above problems.

3.4 Selecting Attributes and Tokenizers

So far we ask an expert user to *manually* select a set of attributes. Then we *concatenate* the values of these attributes into a string, tokenize it *using a default (3-gram) tokenizer*, then index and search on the tokenized string. We call this solution Sparkly Manual.

Sparkly Manual works well, but in certain cases it suffers from three problems. First, it can be difficult even for expert users to select good blocking attributes, e.g., when matching people, an attribute named "phone" may seem good for blocking. But closer inspection shows otherwise, because this attribute stores office phones, not personal phones. Second, concatenating the attributes is problematic because the importance of a token depends on which attribute it appears in, e.g., token "ave" in an address attribute is a stop word because it appears in many addresses, but "ave" in a person-name attribute is informative because it is an uncommon name. Finally, using a single tokenizer is also problematic because different attributes may best benefit from different tokenizers, e.g., using a 3-gram tokenizer for brand names, but a word-level tokenizer for product descriptions.

To address the above problems, we will automatically select blocking attributes and associated tokenizers, as elaborated below.

Problem Definition: First we formally define this selection problem. Let the attributes of tables A and B be $F = \{f_1, \ldots, f_n\}$. Let $T = \{t_1, \ldots, t_m\}$ be a set of tokenizers (e.g., 3-gram, word-level). We define a *configuration* L (or *config* L for short) as a set of (attribute, tokenizer) pairs $L = \{(f_{i1}, t_{i1}), \ldots, (f_{ip}, t_{ip})\}$, where $f_{ij} \in F, t_{ij} \in T, j = 1 \ldots p$. Thus, in a config L different attributes can use different tokenizers.

Let $\mathcal{L}$ be the set of all configs. Our goal is to find the config $L \in \mathcal{L}$ that maximizes the recall. To define recall, we begin by defining the similarity score. Given two tuples $b \in B, a \in A$, we define their similarity score with respect to config L as the sum of the BM25 scores of the individual attributes in the config (tokenized using the assigned tokenizers). Formally, we have

$$s(b, a, L) = \sum_{j=1}^{p} s_j[t_{ij}(b.f_{ij}), t_{ij}(a.f_{ij})].$$

Here $t_{ij}(b.f_{ij})$ applies the tokenizer t_{ij} to the value of attribute f_{ij} of tuple b, producing a bag of tokens, and $t_{ij}(a.f_{ij})$ produces another bag of tokens. Then $s_j[t_{ij}(b.f_{ij}), t_{ij}(a.f_{ij})]$ computes the BM25 score between these two bags of tokens. Note that Lucene uses the above similarity score.

Next, we define the blocking output for when the above similarity score $s(b, a, L)$ is used. For any tuple $b \in B$:

- Let $Q(b, A, k, L)$ be the list of top-k tuples from A that has the highest tf/idf scores, as defined by $s(b, a, L)$, with b, and
- Let $C(b, A, k, L)$ be the set of all pairs (b, v) where we have $v \in Q(b, A, k, L)$.

Then the blocking output is $C(B, A, k, L) = \cup_{b \in B} C(b, A, k, L)$. Finally, let $recall(C(B, A, k, L))$ be the fraction of true matches in $C(B, A, k, L)$. Then our problem is to find a config $L \in \mathcal{L}$ that maximizes $recall(C(B, A, k, L))$ for a given k.

The above problem raises two challenges: how to estimate the recall of a config and how to find the config with the highest recall in a large space of configs. We now address these challenges.

Estimating the Recall of a Config: Given a config L, it is not possible to estimate $recall(C(B, A, k, L))$ because we do not know the true matches. To address this problem, we make the key observation that it is possible to estimate the *discriminative power* of a config L, which captures its ability to tell apart the matches from the non-matches. We can then search for the config with the maximal discriminativeness, on the heuristic assumption that this config is likely to achieve high recall.

Example 3.1. To motivate the notion of discriminativeness, consider a tuple $b \in B$ and three singleton configs L_1, L_2, L_3 involving attributes f_1, f_2, f_3, respectively. Let r_1, r_2, r_3 be the top-k lists for b, produced by querying the inverted index I of table A using the above 3 configs, respectively.

Figure 3.a shows the top-k lists r_1, r_2, r_3. Note that each top-k list contains tuple IDs in A, already sorted in decreasing BM25 scores. For each list, the figure shows the scores, plotted against the ranks of where they appear in the list.

Figure 3.a suggests that for the above tuple $b \in B$, r_3 is quite "discriminative", because it "slopes down" steeply (i.e., the top few tuples of r_3 have very high scores while the rest of the tuples have much lower scores). In fact, the curve r_3 appears more discriminative than r_1 and r_2, which do not "slope down" as much.

It may appear that we can measure this discriminativeness as the area under the curve (AUC): smaller AUC means higher discriminativeness. In Figure 3.a, this is indeed true for r_3 and r_1: $AUC(r_3) <$

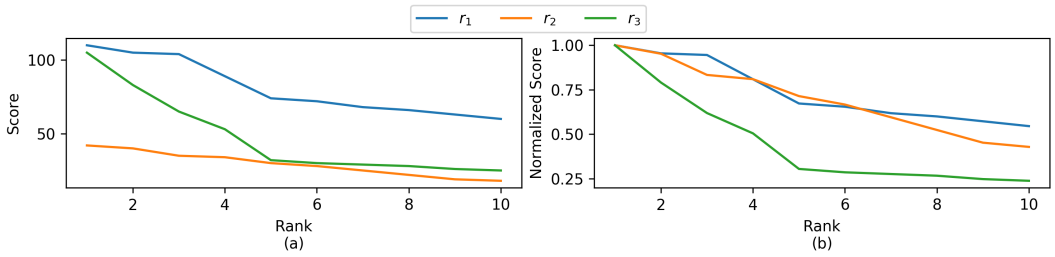

Figure 3: Illustrating the discriminativeness of configs.

$AUC(r_1)$ and r_3 is more discriminative than r_1. But it is not true for r_3 and r_2, because $AUC(r_2) < AUC(r_3)$, yet r_2 is not more discriminative than r_3.

The problem is that the BM25 scores of the curves (generated by using different configs) are not comparable, and hence the AUCs are also not comparable. To address this, we normalize the BM25 scores of each curve to be between [0,1] (by dividing the original scores in each curve by the maximum score of that curve). Figure 3.b shows the normalized curves. Now it is indeed the case that smaller AUC means higher discriminativeness.

Thus, we can define the discriminativeness of a config L for a table B (given a table A and an inverted index I) as the average discriminativeness of config L for each tuple in B: $meanAUC(B, L, k) = \frac{1}{|B|} \sum_{b \in B} AUC(b, L, k)$.

In turn, we can define $AUC(b, L, k)$, the discriminativeness of config L for a tuple $b \in B$, as the normalized AUC. Let $r(b, L, k) = ((v_1, s_1), ...(v_{k'}, s_{k'}))$ be the top-k tuple list retrieved from index I, for record $b \in B$, scored according to config L, sorted in decreasing order of score $s_1, \ldots, s_{k'}$ ($k' \leq k$ because only tuples with positive score can be in the list). Then we can compute the area under the curve as $AUC(b, L, k) = \frac{1}{k' \cdot s_1} \sum_{i=1}^{k'-1} s_{i+1} + \frac{s_i - s_{i+1}}{2}$. The Appendix explains how we arrive at this formula.

In practice, computing $meanAUC(B, L, k)$ for a config L is too expensive, as we have to query index I with all tuples in B. So we approximate it using $meanAUC(B', L, k)$, where B' is a random sample of 10K tuples of B (and we set k to 250).

Searching for a Good Config: Our goal now is to find the config L that maximizes $meanAUC(B', L, k)$. The number of configs can be huge (e.g., in the millions). So we adopt a greedy search approach. First, we score all singleton configs (each using a single attribute/tokenizer pair) and find the top 10 configs with the lowest *meanAUC* scores. Next, we combine these configs to create "composite" configs, where each config has up to 3 attributes. We do not consider configs of more than 3 attributes because in our experience these configs take much longer to run yet only minimally improve recall, if at all. Finally, we score all configs and return the one with the lowest *meanAUC* score.

Since we use at most 10 singleton configs to create configs of size up to 3 attributes, the total number of configs to score is at most 175, making exhaustive scoring of all configs possible, especially because we score the configs in parallel using the Spark cluster.

We further speed up the above search using a technique called *early pruning*. To illustrate, consider again the problem of scoring all singleton configs to find the top 10 configs. Scoring a config means querying the inverted index I with all tuples $b \in B'$. Even though B' is small (currently set to 10K), this still takes time. So we score the configs using a sample B'' which is a small subset of B', use a statistical test to remove all configs for which we can say with high confidence that they will not make it into the top 10, then expand B' with more tuples, re-score the remaining configs, and so on. Specifically:

(1) Initialize the subsample $B'' = \emptyset$ and S to be the set of all configs from which we have to compute the top 10 configs.
(2) Expand the subsample B'' by adding to it a small random sample of h tuples from $B' \setminus B''$.
(3) Compute the *meanAUC* for all configs in S using B'' and finding the set $\hat{R}$ of the top-10 configs.
(4) For each config $L \in S \setminus \hat{R}$, use the Wilcoxon signed-rank test [41] to determine (with high confidence) if its *meanAUC* score is greater than those of the configs in $\hat{R}$. If yes, then L is unlikely to ever be in the top 10. Remove L from S.
(5) If $S = \hat{R}$ or $B'' = B'$, return $\hat{R}$ as the top-10 configs, otherwise go back to Step 2.

We also use the above early pruning procedure to search the space of the larger configs, but we search for top-1 insead of top-10. This means that $\hat{R}$ is a singleton set (as opposed to containing 10 configs) and during Step 4 we drop any config which has estimated *meanAUC* greater than that of the current top-1 in $\hat{R}$.

4 EMPIRICAL EVALUATION

Datasets: We use 15 datasets described in Table 1, which come from diverse domains and sizes, and have been extensively used in recent EM work [24, 27, 36, 40] (except Hospital, which is private). Structured datasets have short atomic attributes such as name, age, city. Textual datasets have only 2-3 attributes that are textual blobs (e.g., title, description). For dirty EM, we focus on one type of dirtiness, which is widespread in practice [27] mainly due to information extraction glitches, where attribute values are "moved" into other attributes (e.g., the value of "brand" is missing and appears in attribute "title"). Textual and dirty datasets are derived from the corresponding structured datasets (e.g., the textual dataset Amazon-Google$_2$ is derived from the structured dataset Amazon-Google$_1$).

Later we use 6 additional datasets for certain experiments, as discussed in Section 4.5 and Section 5.

Methods: We compare Sparkly to 8 state-of-the-art (SOTA) EM blockers.

Type	Dataset	Table A	Table B	#Matches	#Attr
Structured	Amazon-Google$_1$	1,363	3,226	1,300	4
	Walmart-Amazon$_1$	2,554	22,074	1,154	6
	DBLP-Google$_1$	2,616	64,263	5,347	4
	DBLP-ACM$_1$	2,616	2,294	2,224	4
	Hospital$_1$	1,786	1,786	3,949	7
	Songs-Songs$_1$	1,000,000	1,000,000	1,292,023	5
Textual	Amazon-Google$_2$	1363	3,226	1,300	2
	Walmart-Amazon$_2$	2,554	22,074	1,154	2
	Abt-Buy	1,081	1,092	1,097	3
Dirty	Amazon-Google$_3$	1,363	3,226	1,300	4
	Walmart-Amazon$_3$	2,554	22,074	1,154	6
	DBLP-Google$_2$	2,616	64,263	5,347	4
	DBLP-ACM$_2$	2,616	2,294	2,224	4
	Hospital$_2$	1,786	1,786	3,949	7
	Songs-Songs$_2$	1,000,000	1,000,000	1,292,023	5

Table 1: Datasets for our experiments.

Autoencoder, Hybrid, Union(DL,RBB): A recent work [40] shows that deep learning (DL) based blockers significantly outperform many other blockers. So we compare Sparkly to the two best DL blockers: Autoencoder and Hybrid [40]. The work [40] also shows that combining the best DL blocker and RBB, a SOTA industrial blocker, produces even better recall at a minimal increase of blocking output size. As a result, we also compare Sparkly with that blocker, henceforth called Union(DL,RBB).

PBW, DBW, JD: Hash blockers have been very popular, and recent EM work has developed highly effective hash blockers, as captured in the pioneering JedAI open-source EM platform [32, 34]. These blockers hash each tuple to be matched into multiple blocks, one per each unique token in the tuple, then employ sophisticated methods to remove/clean blocks, among others. Based on personal communications with the JedAI authors, we compare Sparkly to 3 SOTA blockers in JedAI: PBW, DBW, and JD (the Appendix describe these methods).

kNN-cosine, kNN-jaccard: Finally, a recent work [36] shows that a kNN blocker outperforms many other blockers. This blocker finds all tuple pairs where a tuple is among the k nearest, i.e., most similar, neighbors of the other tuple, where the similarity measure is cosine over 5-gram tokenization. So we compare Sparkly with this blocker, denoted KNN-cosine. We also compare Sparkly to kNN-blockers where the similarity measure is cosine over 3-gram tokenization and Jaccard over 3-gram and 5-gram tokenization.

4.1 Recall and Output Size

We begin by comparing Sparkly to existing methods in terms of recall and output size. To keep the comparison manage-able, we first compare SM, the Sparkly version where the user manually selects the attributes to be blocked on (i.e., Sparkly Manual), with all SOTA blockers. Then we compare SM with SA, the Sparkly version that automatically selects blocking attributes (i.e., Sparkly Auto).

Comparing SM to DL Methods: Figure 4 compares SM with the two best DL blockers, Autoencoder and Hybrid [40]. The figure shows 15 plots, one per dataset. Consider the first plot, which is for the structured Amazon-Google dataset. Here the x-axis shows *the recall* $R = |C \cap G|/|G|$, where C is the blocking output and G is the set of all gold matches. The y-axis shows *the candidate set size ratio CSSR* $= |C|/|A \times B|$. Both axes show values *in percentage*. So a value of 85 on the x-axis means recall of 85%, and a value of 5 on the y-axis means CSSR of 5%. Like SM, the two DL blockers Autoencoder and Hybrid are also top-k. So we vary the value of k to generate the above plot. We generate the remaining 14 plots in a similar way. Note that the y-axes of the 15 plots vary significantly in scale. This is necessary so that we can show the difference among the curves.

All 15 plots show that SM significantly outperforms the two DL blockers: for each recall value, SM achieves a much lower CSSR, and this gap widens dramatically as recall approaches 100%. These gaps are bigger for textual datasets, suggesting that SM can better handle textual data than the DL blockers. The gaps are smaller but still quite significant on all dirty datasets.

The above two DL methods concatenate all attributes and then block on the concatenation. In the next experiment, we modified them to block on *the concatenation of only those attributes that SM blocks on*. Figure 5 shows the results. Even in this case, SM still outperforms both DL methods on 14 datasets (sometimes by very large margins) and is comparable on 1 dataset.

Comparing SM to Union(DL,RBB) and JedAI Methods: Next we compare SM with Union(DL,RBB), which combines the best DL blocker and RBB (a SOTA industrial blocker), and the three JedAI methods: PBW, DBW, and JD. It is very difficult to vary the parameters of these methods in such a way that generates meaningful recall-CSSR curves, because they do not have a top-k parameter that we can adjust. So we compare them with SM at $k = 10, 20, 50$, as shown in Table 2.

This table shows that SM is very predictable: it achieves high recall for all datasets (92.5-100% for $k = 10$, 96.4-100% for $k = 20$, 98.7-100% for $k = 50$), and its output size is capped as $k * |B|$. In contrast, the remaining four methods are unpredictable. For example, PBW's recall can be perfect (100%) but also can be as low as 74.5%, and its output size can be small but can also be as high as 4.2 billions for the structured dataset Songs. (We report no results for "S - D" because PBW was out of memory on this dataset, on a machine with more than 100G of RAM). Similarly, DBW's recall can be as low as 84.7% and output size as high as 454.5M.

JD produces much more reasonable output size across all datasets, but at the cost of lower recall 35.4-96.4%. Similar to JD, Union(DL,RBB) also produces reasonable output sizes (larger than those of JD), but varying recalls 83-99.9%.

Comparing SM to kNN Methods: Finally, we compare SM with the kNN methods. As mentioned earlier, a recent paper [36] finds that kNN-cosine using 5-gram tokenization outperforms many SOTA blockers. So we compare SM with this method. We also compare SM with kNN using Jaccard (with 3-gram tokenization).

Figure 6 shows the recall and blocking output size of these methods. Note that we also evaluated kNN-cosine with 3-gram tokenization, but will not discuss it any further because we found its performance to be comparable to kNN-cosine with 5-gram tokenization).

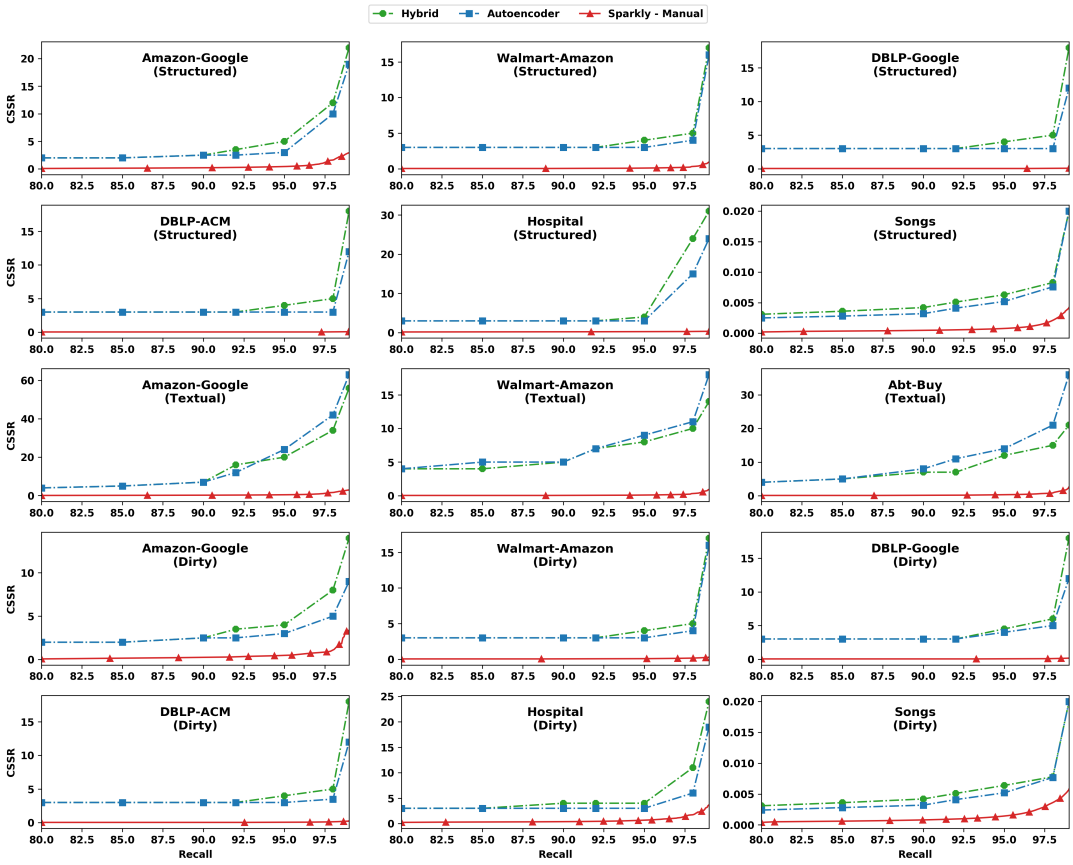

Figure 4: SM vs. the two best DL methods in terms of recall and blocking output size.

Dataset	PBW		DBW		JD		Union (DL,RBB)		Sparkly K=10		Sparkly K=20		Sparkly K=50	
	$\|C\|$	Recall	$\|C\|$	Recall	$\|C\|$	Recall	$\|C\|$	Recall	$\|C\|$	Recall	$\|C\|$	Recall	$\|C\|$	Recall
AG - S	24.5k	92.1	15.9k	89.2	5.9k	80.5	77.7k	98.8	33.3k	96.8	66.5k	97.8	165.9k	99.2
WA - S	1.5m	99.7	159.8k	93.8	88.3k	95.0	2.1m	98.9	220.7k	98.4	441.4k	99.0	1.1m	99.5
DG - S	430.5k	91.0	779.3k	99.6	53.1k	79.7	7.6m	99.6	641.1k	99.9	1.3m	100.0	3.2m	100.0
DA - S	8.1k	83.7	35.1k	99.9	2.3k	80.3	198.4k	99.9	22.9k	99.8	45.9k	100.0	114.7k	100.0
H - S	11.9k	100.0	4.0k	84.7	1.4k	35.4	209.8k	99.9	17.8k	100.0	35.4k	100.0	85.4k	100.0
S - S	4.2b	100.0	379.4m	99.8	2.5m	82.0	50m	98.7	10.0m	96.3	20.0m	97.9	50.0m	99.3
AG - T	24.5k	92.1	15.9k	89.2	5.9k	80.5	33.6k	85.0	33.3k	96.8	66.5k	97.8	165.9k	99.2
WA - T	1.5m	99.7	159.8k	93.8	88.3k	95.0	7.9m	83.0	220.7k	98.4	441.4k	99.0	1.1m	99.5
AB - T	4.7k	74.5	6.0k	88.6	1.2k	65.2	44.6k	95.7	10.9k	98.1	21.8k	98.9	54.5k	99.2
AG - D	38.8k	94.1	18.7k	91.3	6.4k	79.5	360.0k	99.3	33.3k	96.6	66.5k	98.2	166.0k	99.0
WA - D	1.1m	99.5	225.2k	97.4	88.1k	95.9	935.9k	97.9	220.7k	99.1	441.5k	99.7	1.1m	99.8
DG - D	4.0m	99.7	925.5k	98.8	180.5k	96.4	47.6m	99.8	642.2k	99.9	1.3m	100.0	3.2m	100.0
DA - D	12.5k	86.6	42.0k	97.2	4.7k	82.4	1.0m	99.8	22.9k	99.3	45.9k	99.8	114.7k	100.0
H - D	22.5k	100.0	31.2k	87.9	2.4k	56.1	136.8k	98.5	17.9k	94.0	35.6k	97.1	88.4k	98.7
S - D	—	—	454.5m	96.2	3.1m	68.3	50m	95.2	10.0m	92.5	20.0m	96.4	50.0m	98.8

Table 2: SM vs. the three JedAI methods and Union(DL,RBB) in terms of recall and blocking output size.

When comparing kNN-Jaccard to SM, Figure 6 shows that there are four datasets where kNN-Jaccard achieves comparable results to SM: DBLP-ACM, Hospital, Songs, and DBLP-Google (all structured). For the textual datasets, SM is consistently better than kNN-Jaccard by a small margin, achieving lower CSSR for the same recall at all points on the curve. The story is very different when we examine the

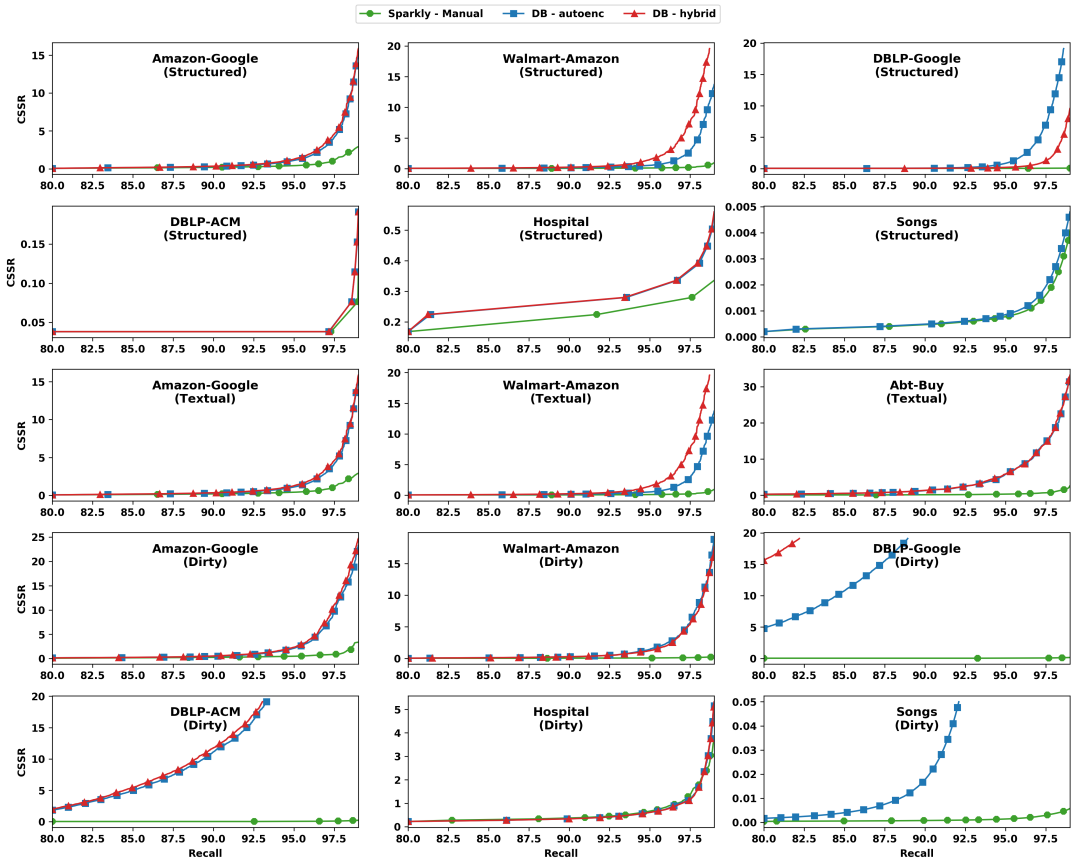

Figure 5: Comparing SM to DL methods which use the same attributes as SM to block; here "DB - autoenc" and "DB - hybrid" refer to the Autoencoder and Hybrid methods of the DeepBlocker paper [40], respectively.

dirty datasets, where SM consistently out performs kNN-Jaccard, in some cases by a very large margin.

For structured datasets, kNN-cosine achieves comparable results to SM on Songs, DBLP-Google, and DBLP-ACM, while underperforming on Hospital structured. For textual datasets, kNN-cosine achieves similar results to kNN-Jaccard overall, doing better on Amazon-Google but worse on Abt-Buy, making SM overall the best on the textual datasets. For dirty datasets, SM does much better than kNN-cosine, in some cases by a large margin.

For the structured and textual datasets, the attributes used for blocking are "information dense" and have relatively small variance in terms of length. These characteristics are important for two reasons. First, since the attributes were information dense they contained few if any repeated tokens. This implies that the term frequency information used by BM25 is unlikely to play a significant role in SM's superior performance over kNN-Jaccard and kNN-cosine. Second, since most attributes within a dataset have similar length, the document length normalization is also unlikely to play a significant role in SM's performance. Together these two points suggest that the IDF weighting is an important factor for SM's outperforming kNN-Jaccard and kNN-cosine on the structured and textual datasets.

Turning our attention to the dirty datasets, we see that the performance gap between SM and the kNN methods is much larger. We attribute this to two factors. First, the dirty datasets contain significantly more noise in the attributes used for blocking. It is likely that the IDF weighting used by BM25 is an effective means of down weighting the noisy tokens present in the dirty datasets. Second, the length of the attributes, and hence the sizes of the bags of tokens, varies far more in the dirty datasets than in the structured or textual datasets. This suggests that the length normalization of BM25 is far more effective at dealing with these kinds of variations as opposed to the length normalization of kNN-cosine and kNN-Jaccard.

To summarize, we find that on 10 datasets SM outperforms kNN-cosine-5gram, sometimes by huge margins. On 1 dataset SM is comparable, and on the remaining 4 datasets SM is worse than kNN-cosine-5gram, but the performance gap is very small. We also find that kNN-cosine using 3grams is comparable to kNN-cosine using 5grams, and both outperform kNN using Jaccard (either 3grams or 5grams).

Comparing SM to SA: Figure 7 shows that SA outperforms SM on 10 datasets in terms of recall and output size, sometimes by a large margin. SA is worse than SM on the remaining 5 datasets, but

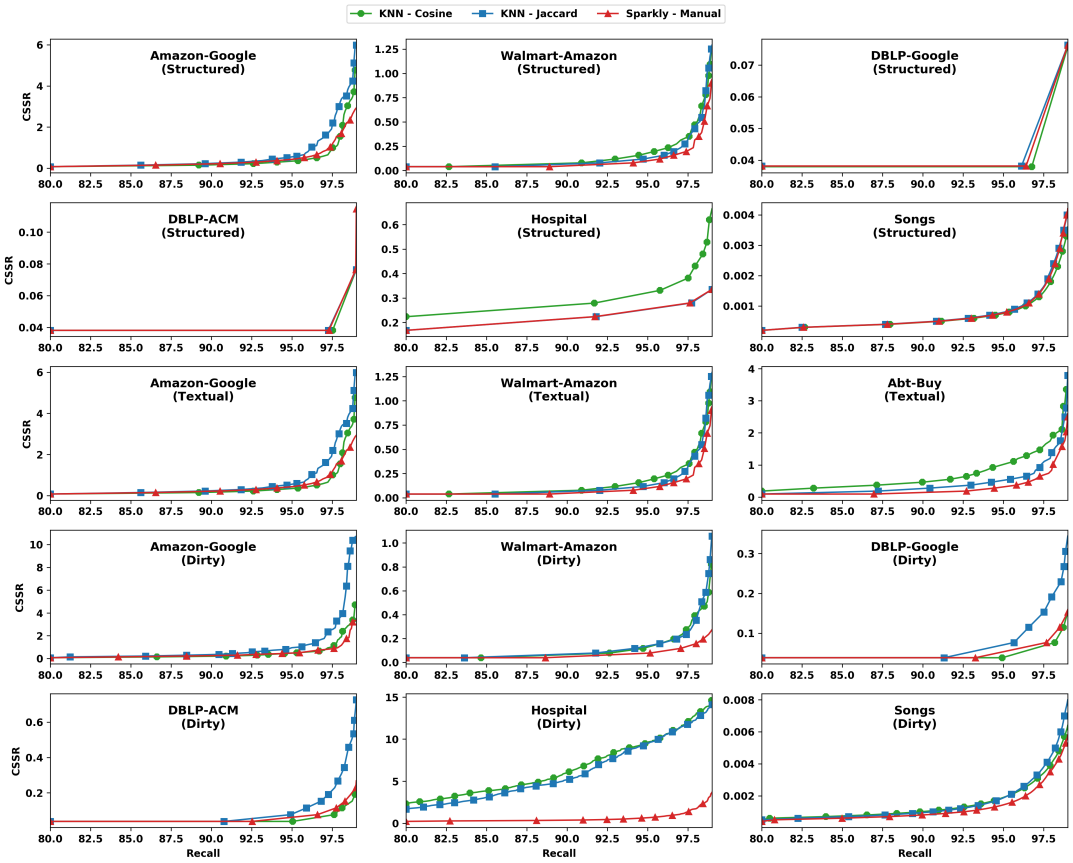

Figure 6: SM vs. kNN-Cosine (5-gram) and kNN-Jaccard (3-gram) in terms of recall and blocking output size.

the performance gap is very small. For example, at 98% recall, SA's output size is at most 0.7% larger than that of SM.

We are still examining why SA is worse than SM on some datasets. But as far as we can tell, it appears that in some cases SA adds a new attribute X to the set of blocking attributes, and X has many missing values. This negatively affects SA's accuracy. Overall, we found that dealing with missing values has been difficult, and more work is needed in this aspect (for both blocking and matching).

4.2 Runtime of Sparkly

We now examine the runtime of Sparkly. We ran all experiments on an AWS cluster of 10 nodes. Each node is an m5.4xlarge instance with 16 cores, 64G RAM, costing $0.75/hour (as of July 2022).

Figure 8.a shows the runtime of SM (the solid lines) and SA (the dotted lines), as we vary the size of two datasets: WDC and Songs. WDC has 26M tuples [37] and Songs has 1M tuples (we cannot use WDC for recall experiments because it does not have all gold matches). A point n on the x-axis reports the runtimes measured on a sample of n millions tuples randomly obtained from WDC, and on a sample of n millions tuples obtained by replicating Songs n times.

The above figure shows that both SM and SA scale (slightly superlinearly) as we increase the dataset size, and that *SA is much faster than SM*. This is because SA scores the attributes individually, instead of scoring their concatenation as SM. Concatenation produces long attributes, and performing top-k search on long attributes takes more time than on short attributes. Further, SA can use word tokenizers on some attributes, whereas SM only uses 3gram tokenizers. This produces fewer tokens, which often leads to faster top-k search.

It is noteworthy that SA can block datasets of size 10M under 100 minutes, incurring an AWS cost of only $12.5.

Figure 8.b shows the runtime of Sparkly on the 10M WDC and 10M Songs datasets, as we vary the size of the AWS cluster. As the number of nodes goes from 3 to 15, runtime decreases significantly, as expected. As we increase the cluster size, eventually the overhead time (indexing, shipping, attribute/tokenizer selection, etc.) will dominate (compared to the top-k probing time).

We discuss the runtime of existing SOTA methods in Section 4.5.

4.3 Performance of Sparkly's Components

We find the indexing time to be minimal. For example, on the same AWS cluster described above, indexing Songs at size 5M and 10M takes 76 and 115 seconds, respectively. The resulting index sizes

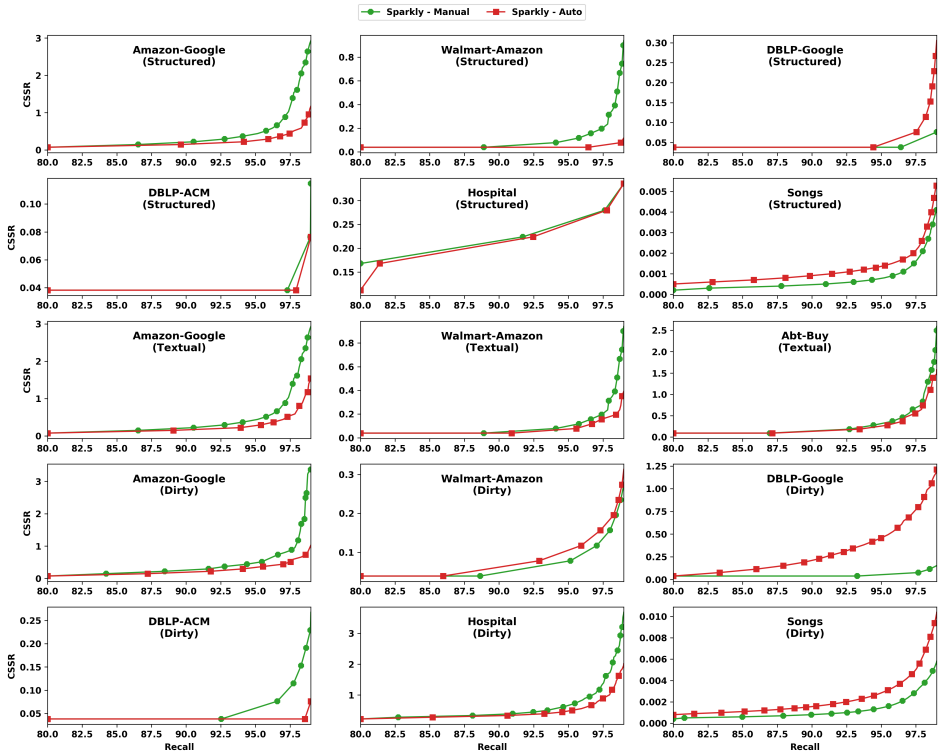

Figure 7: SM vs. SA in terms of recall and output size.

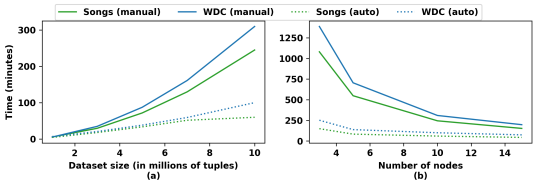

Figure 8: Runtime for (a) varying dataset sizes, and (b) varying cluster sizes, using datasets of 10M tuples each.

are also reasonable. For Songs and WDC at size 1M, 5M, and 10M, the sizes are 137, 664, 1318 MB, and 214, 1034, and 2042 MB, respectively. Shipping these indexes to the Spark nodes takes minimal time. For example, for Songs and WDC at 1M, 5M, and 10M, shipping the indexes takes 2.2, 7.2, 21 seconds, and 2.5, 11, 32 seconds, respectively.

Finally, recall that SA performs a search for a good set of attributes/tokenizers (to block on). On Songs and WDC 1M, 5M, and 10M, *without early pruning*, this search takes 4, 9.2, 15.6 mins and 4.6, 10.1, 17.2 mins, respectively. Early pruning cuts these times by up to 70%, to 1.2, 3.2, 6 mins, and 2, 5.3, 14 mins, respectively.

The greedy method used by the searcher was quite effective. We performed exhaustive search on 11 datasets to find the optimal configs, and found that the greedy method found a config with a score within 0-0.8% of the optimal score on 10 datasets and within 10% on 1 dataset.

4.4 Sensitivity Analysis

We now vary the parameters of the major components to examine the sensitivity of Sparkly.

Blocking Attributes: Recall that in SM, the user manually selects a set of attributes S to block on. Figure 9 examines varying S. Consider the first plot, Amazon-Google. Here SM blocks on attribute "title", producing the curve in red. Then the plot also shows the curves where SM blocks on "title" plus another attribute. The remaining 14 plots are structured similarly. (Some curves are under other curves and thus are not visible.)

Figure 9 shows that varying the set of blocking attributes does impact the performance of SM, minimally by 0-1.5% CSSR in 11 datasets, and moderately by 2-5% CSSR in 4 datasets. This suggests that while manually selecting blocking attributes is a reasonable strategy, there is still room to improve, e.g., by automatically finding such attributes, as done in SA.

Tokenizers: Recall that SM uses a 3gram tokenizer. Next we examine replacing this tokenizer with a 2gram, 4gram, and word-based tokenizer, respectively. Figure 10 shows the results. We find that changing the tokenizer can significantly impact the performance (e.g., by up to 11.5% CSSR in the first plot). Overall, the 3gram tokenizer (used by SM) is a good choice as it has reasonable performance on most datasets. The 2gram and 4gram tokenizers perform worst, with the 2gram tokenizer also incurring the longest runtime.

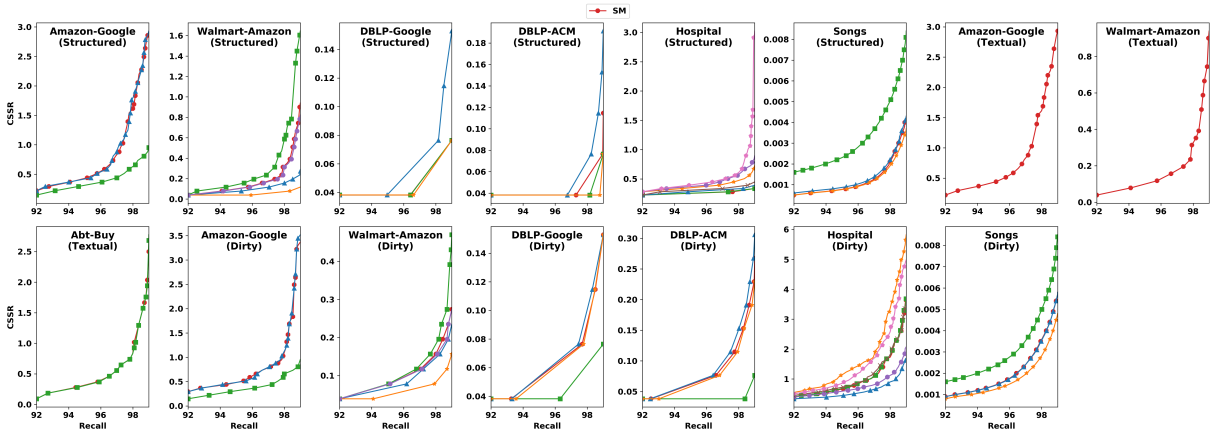

Figure 9: Performance of SM for varying sets of blocking attributes.

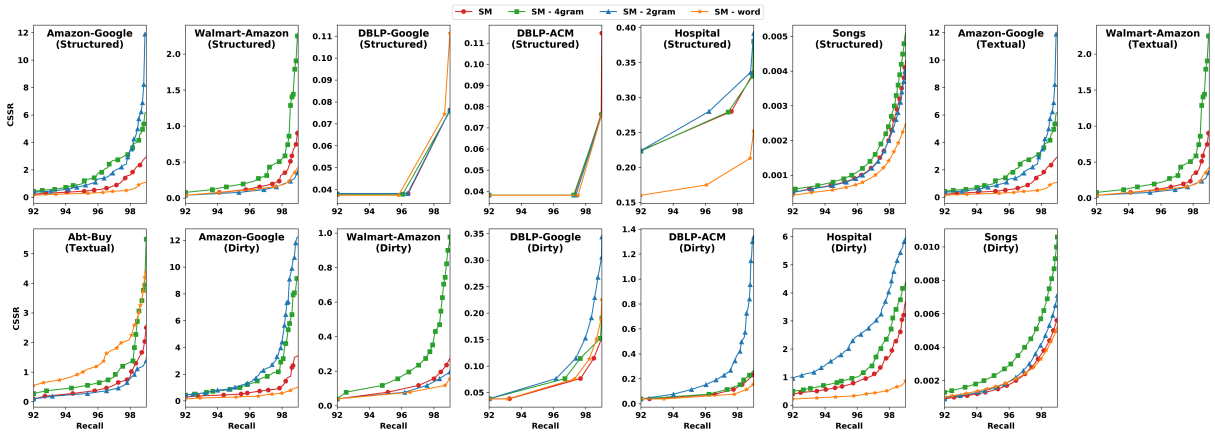

Figure 10: Performance of SM for different tokenizers.

BM25's Parameters: BM25 has two parameters: k_1 (default value 1.2) and b (default value 0.75), to handle term saturation and document length. Varying k_1 from 1 to 2 does not significantly change SM's performance. This may be because in our setting blocking attributes do not have many terms, and they do not have high term frequency, so term saturation is not a major issue. Varying b from 0.5 to 1 changes SM's performance more, by up to 2% CSSR. But we also find that $b = 0.75$ provides a good default value for SM, as its curve is either the best curve or very close to the best curve on most datasets. See the Appendix for details.

Config Searcher's Parameters: Recall that SA uses a searcher to find a good blocking config. This searcher has four major parameters: (1) the size of B', a sample of table B on which to score the configs (set to 10K), (2) the number of tuples returned in each querying $k = 250$, (3) the number of initial configs selected (set to 10), and (4) the max number of attributes considered in a config (set to 3).

Varying (1) from 5K to 15K changes SA minimally. Varying (2) from 200 to 300 again changes SA minimally (only up to 0.2% CSSR on 1 dataset). Similarly, varying (3) from 8 to 12 and varying (4) from 2 to 4 show minimal changes. In all cases, the default values for (1)-(4) provide a good curve, which is either the best or very near the best (see the Appendix for details).

4.5 Additional Experiments

We now examine how Sparkly performs on very large datasets, how it compares runtime-wise to DL methods, and whether DL methods can achieve higher accuracy, given larger datasets (to train on).

It is very difficult to find very large *public* datasets with *complete gold*, i.e., all true matches (without which we cannot compute the blocking recall). After an extensive search, we settle on three datasets: BC, MB, and WDC. BC (Big Citations) blocks two tables of 2.5M and 1.8M paper citations. MB (Music Brainz) blocks a table of 20M songs (against itself), and WDC blocks a table of 26M product descriptions [37]. BC and MB have complete gold, but WDC does not (see the Appendix for a description of these datasets).

Table 11 shows the results. First, we deployed an AWS cluster of 30 m5.4xlarge nodes (16 cores, 64G RAM, $0.75/hour, per node), then ran Sparkly on all three datasets (see the first three rows of the table). Each row lists the results of SM and SA separated by "/".

Method	Dataset	Time	Recall @ 10	Recall @ 25	Recall @ 50
Sparkly	WDC 26M	603/130	-	-	-
	MB 20M	449/168	79/95	87/97	91/98
	BC 2.5M	44/11	99/79	100/89	100/94
	MB 10M	132/61	85/96	91/98	94/98
Autoencoder	WDC 10M	925	-	-	-
	MB 10M	691	30	35	40
	BC 2.5M	146	81	84	85
Hybrid	BC 2.5M	2719	73	76	78

Figure 11: Applying Sparkly and DL methods to large datasets.

Column "Time" shows the total time in minutes, while the next three columns show the recall at $k = 10, 25, 50$. We cannot compute recall for WDC as it does not have the complete gold.

The first three rows show that *Sparkly scales to very large datasets, and that SA is much faster than SM,* taking only 130 and 168 mins to block WDC 26M and MB 20M, respectively, at $k = 50$ and at a reasonable cost of less than $67.5 on AWS. Sparkly achieves high recall on these datasets at $k = 50$.

Skipping the 4th row of Table 11 (which we discuss later), we now consider the DL method Autoencoder. Unfortunately we had tremendous difficulties scaling Autoencoder to large datasets. Autoencoder is a prototype code used in the paper [40], for datasets of up to 1M tuples. It runs on a single GPU and uses many Python libraries that are not well suited to large datasets (e.g., the SVD implementation of Sklearn). So when applied to large datasets, Autoencoder quickly exhausts all memory and crashes, and there is no easy way to modify it to run in a distributed setting (where it can use a lot more GPU memory).

After intensive optimization efforts, we managed to apply Autoencoder to BC, WDC 10M, and MB 10M, on a SOTA hardware available to us (32t/16c CPU with 64G RAM coupled with RTX 2080ti GPU with 11G RAM). Table 11 shows the results. While it is not entirely fair to compare the runtimes of Autoencoder and Sparkly, because they run on *different* hardware, it is still interesting to note that Autoencoder takes much more time than Sparkly, e.g., 691 vs 132/61 mins (for SM/SA) on MB 10M, and 146 vs 44/11 mins on BC 2.5M. Autoencoder spent most time in preprocessing and self-supervised training.

Hybrid is far more complex than Autoencoder, and we only managed to run it on BC 2.5M (it ran out of memory on WDC 5M and MB 5M). Even on BC, its runtime is already very high (2719 mins). This suggests that *existing prototype DL blockers do not scale to large datasets, requiring a lot more future work on this topic.*

Both Autoencoder and Hybrid achieve far lower recall at $k = 50$ than Sparkly (see the rows for BC 2.5M and MB 10M), suggesting that *these methods still cannot exploit larger datasets to achieve higher accuracy than Sparkly.* In the Appendix we discuss how the remaining SOTA methods also do not scale to these large datasets.

5 DISCUSSION & FUTURE WORK

We now discuss questions that may arise in light of Sparkly's strong blocking performance.

Why little attention so far? TF/IDF measures have long been studied in the matching step of EM [9]. It is not clear why they have received no attention in the blocking step (except several works as discussed in Section 6). A possible reason is that so far EM researchers have preferred to focus on hash-based blocking, which is conceptually simple and easy to scale [6, 30, 33]. Even when considering similarity-based blocking, researchers have preferred similarity measures that seem simpler and more amenable to scaling, e.g., edit distance, Jaccard, overlap, cosine [6, 30, 33]. TF/IDF appears difficult to scale. A straightforward application of inverted indexes is slow, and it was not obvious how to do much better.

Up until 2015, Lucene was also slow, making tf/idf blocking using Lucene impractical. Then it adopted the block-max WAND indexing technique and became much faster [19]. As this paper has shown, a combination of the "new" Lucene and Spark has now made tf/idf blocking very competitive, and suggests that going forward it should receive more attention.

Top-k vs Thresholding: We believe doing top-k search is critical. To see this, we compute the similarity scores of *all gold matches* on each dataset, for tf/idf, Jaccard, and cosine measures.

Figure 12 shows the kernel density estimation (KDE) curves of these scores, on all 15 datasets. They can be read like smoothed histograms (we do not show the traditional histograms of these scores as there are too many to show per plot).

Essentially, each curve can be read like a probability distribution curve (e.g. a normal curve). For example, consider the area under the curve for the interval [0.6, 0.8] for Jaccard-3gram in the Amazon-Google Structured plot (top left). Let this quantity be $AUC(0.6, 0.8)$. The (estimated) probability that a gold match, taken randomly from all gold matches, will have a Jaccard-3gram score in the interval [0.6, 0.8] is $AUC(0.6, 0.8)/AUC(0.0, 1.0)$, i.e., the area under the curve of the interval divided by the total area under the curve.

Figure 12 shows that only on 4 datasets would most of these scores be in a narrow range near 1.0 (e.g., between 0.8 and 1), making thresholding work well, i.e., achieving high recall. In the remaining 11 datasets, these scores are spread all over the range [0,1]. This suggests that thresholding (i.e., retain only those tuple pairs whose similarity score exceeds a pre-specified threshold) cannot achieve high recall, unless we set the threshold very low, which blows up the blocking output size.

In contrast, Section 4 shows that doing top-k (as in Sparkly) achieves high recall without blowing up the output size. Thus, we believe that *future blocking solutions should seriously consider doing top-k, rather than thresholding.*

Do We Need TF/IDF? SM and SA, which use tf/idf, outperforms methods that do not, such as kNN-consine, kNN-jaccard, etc. This is most likely because tf/idf can "down weigh" common tokens. For example, the Hospital Dirty dataset has many names such as "David Smith MD Physician" and "Julie Smith MD Physician". When blocking on these names, tf/idf can "ignore" the common tokens "MD", "Physician", whereas standard Jaccard and cosine measures cannot.

So it seems that we do need tf/idf. But do we need both for blocking? To answer this question, we performed several experiments. Figure 13 shows the results when we remove IDF from SM (by dropping "idf(t)" in the BM25 formula of Equation 2 in Section 3). The figure shows that SM greatly outperforms SM-no-idf.

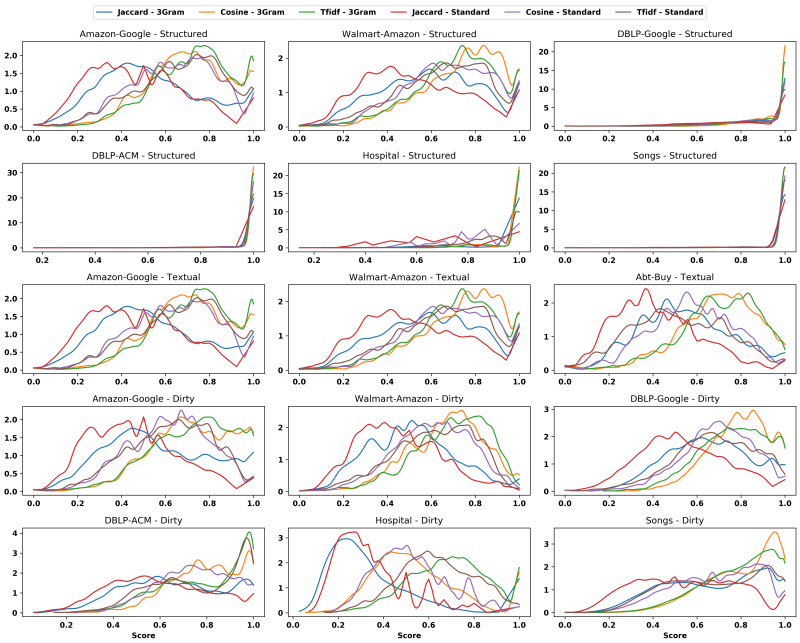

Figure 12: Distributions of the scores of gold matches.

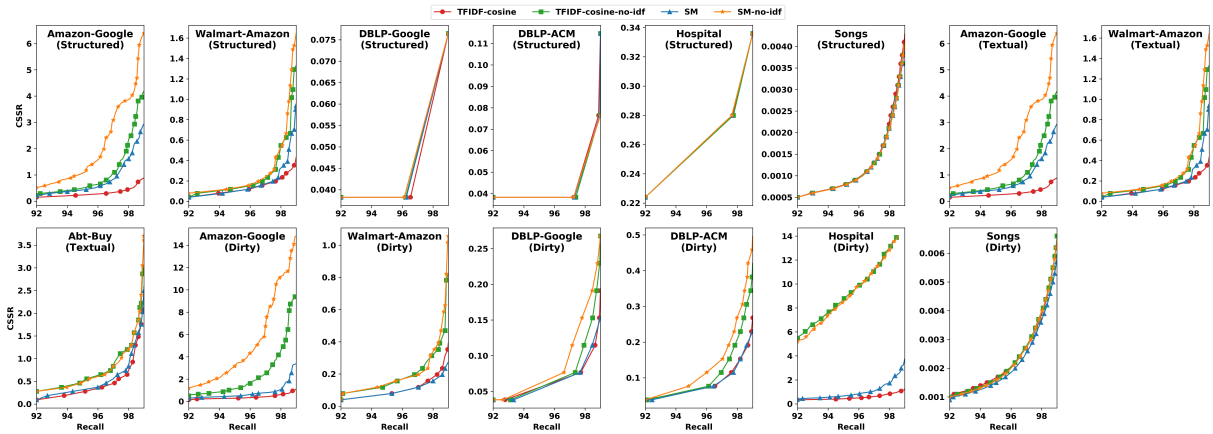

Figure 13: The effect of removing IDF from SM and TFIDF-cosine.

Similarly, when we remove idf from TFIDF-cosine, by dropping "idf(t)" from the formula $V_D(t) = tf(t, D) \cdot idf(t)$ (see Section 3, recall that TFIDF-cosine is the well-known tf/idf measure that computes $s(D, Q) = [\sum_t V_D(t) \cdot V_Q(t)]/[\sqrt{\sum_t V_D(t)^2} \cdot \sqrt{\sum_t V_Q(t)^2}]$, the figure shows that TFIDF-cosine greatly outperforms TFIDF-cosine-no-idf on many datasets. This suggests that IDF is important for blocking.

Interestingly, when we performed a similar experiment where we removed TF, we did not see a clear trend. See Figure 14. TFIDF-cosine and TFIDF-cosine-no-tf perform largely the same. SM is minimally better than SM-no-tf on some datasets, minimally worse on some others, and about the same on the remaining datasets.

So TF seems to have minimal effect on the 15 datasets. We believe this is because the attributes to block on (e.g., product title, person name) tend to be short, where few tokens repeat multiple times. To test this hypothesis, we consider Companies, a highly textual dataset where each tuple is a long document describing a company, and we need to block using the entire tuple (this dataset was used in the paper [27] to evaluate DL methods for matching, see the Appendix for a discussion). Indeed on this dataset where many tokens repeat multiple times, SM greatly outperforms SM-no-tf, e.g., achieving 62% vs 33% recall at $k = 50$. Similarly, TFIDF-cosine greatly outperforms TFIDF-cosine-no-tf. This result suggests that TF's effect on short blocking attributes is minimal, but can be significant on long textual blocking attributes.

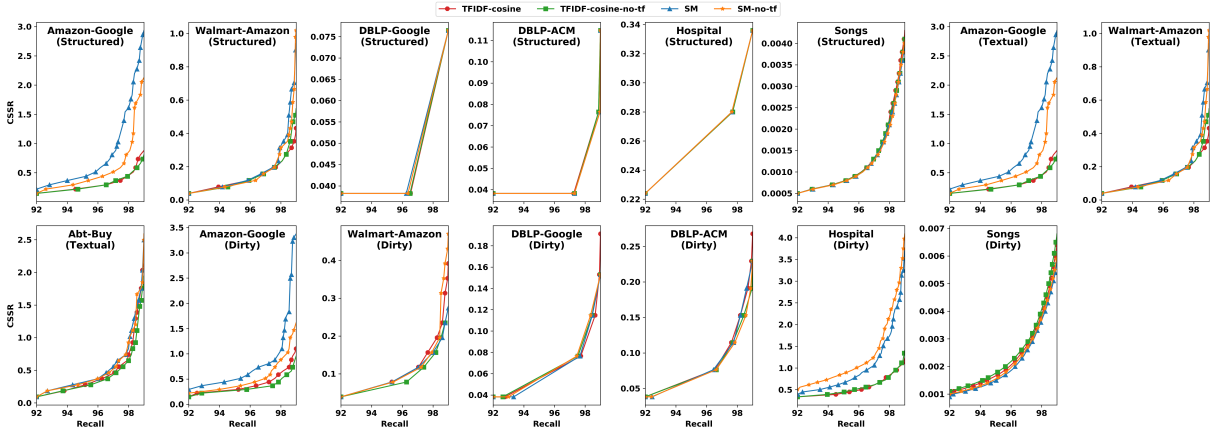

Figure 14: The effect of removing TF from SM and TFIDF-cosine.

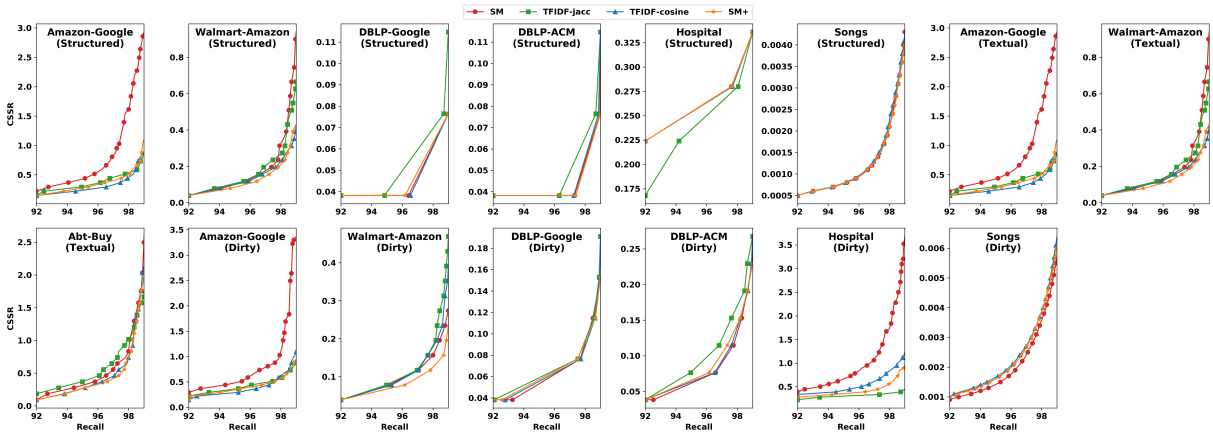

Figure 15: Evaluating different scoring functions.

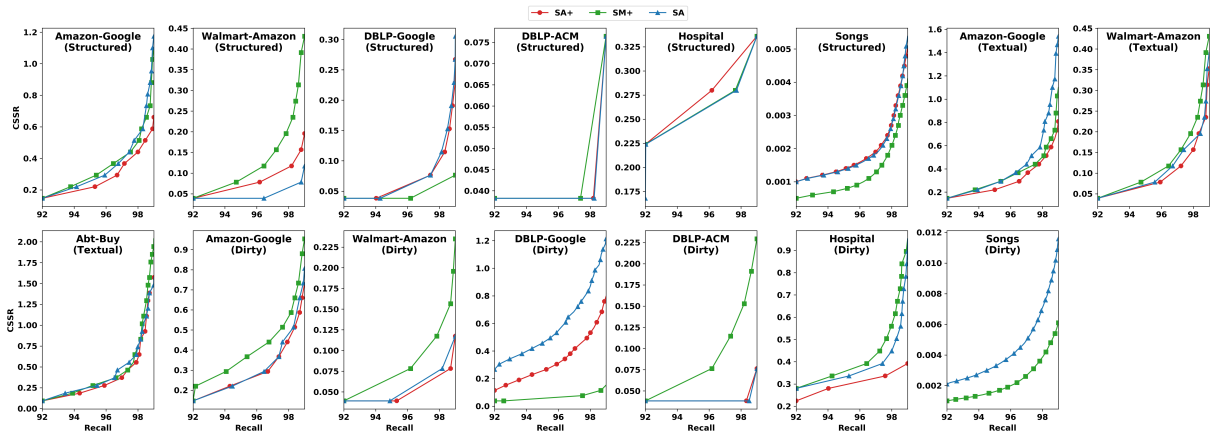

Figure 16: Comparing SM+, SA, and SA+.

Which Scoring Functions are Best? The above results suggest using both TF and IDF. But which scoring function works best? To explore, we examine four scoring functions: TFIDF-cosine, TFIDF-jacc, SM, and SM+. TFIDF-cosine is the cosine similarity function using TF/IDF, described earlier, while TFIDF-jacc is the function fms^{apx} described in [5], which can be viewed as the Jaccard similarity function using IDF. SM+ is an extension of SM that we will describe shortly.

Figure 15 shows that TFIDF-jacc is somewhat worse than TFIDF-cosine. But surprisingly, TFIDF-cosine is better than SM on many datasets. We found this is because TFIDF-cosine's scoring function incorporates the TF and IDF of each term from both the query Q and the document D sides (see Equation 1 in Section 3), but the BM25 scoring function used by SM does not. It incorporates only TF and IDF of each term from the document D's side. In other words, *TFIDF-cosine treats Q and D uniformly, whereas SM does not.* This makes sense in keyword search, where typically Q has few terms, each occurring only once and all terms in Q are important. But these are not true in EM, where Q is a tuple in Table B (and is often as long as D, which is a tuple in Table A). Here, it makes more sense to treat Q and D uniformly, like TFIDF-cosine.

Using the above observation, we modify BM25 to incorporate TF and IDF from Q's side (see the Appendix for details), producing the SM+ solution. Figure 15 shows that SM+ performs very well, being either the best solution or very close to the best solution on all datasets.

We modified SA similarly to produce the SA+ solution. Figure 16 compares SM+, SA, and SA+. We found that SA+ outperforms SA: better or equal to SA in 12 datasets, and worse in 3 datasets. In general, SA+ is either the best solution or very close to the best solution in 14 datasets.

In general, we believe that a good scoring function should not just enable high recall, but also other desirable capabilities, such as fast incremental index updates (as we add/remove tuples, see more below). In addition, it would be even better if there are already open-source software implementing the scoring function and enabling fast blocking on large amounts of data (such software can take years to build).

We note that BM25 (and the modified BM25 as described earlier) meets these criteria. They provide high recall and enable fast incremental index update. There is also a ready-to-use open-source software in the Lucene-based Sparkly system. In contrast, TFIDF-cosine does not enable fast incremental index update, and we are not aware of any open-source software using TFIDF-cosine that can perform fast blocking for large data (e.g., tens of millions of tuples).

All these results suggest that *using tf/idf weighting, and the BM25 scoring function, but modifying it to incorporate TF and IDF from the query side is a promising future direction to explore.*

Qualitative Error Analysis: We have performed a preliminary qualitative error analysis to gain insights into why Sparkly does better than SOTA solutions. For example, when examining tuple pairs that Sparkly correctly declare matches but DL methods do not (by this we mean Sparkly finds these pairs given low top-k, whereas DL methods need a much higher k value to find these pairs), we found that for these tuple pairs, the lengths of the names tend to differ significantly, and the Jaccard 3gram scores are very low. This is potentially important because the FastText package used by DL methods uses q-grams to generate the embeddings.

Another interesting pattern is that these tuple pairs often have a unique product code. Examples include (sony str-de197 av receiver 100 watts per channel, sony stereo audio receiver strde197) and (olympus b-90su aa/aaa size battery charger, olympus olympus ni-mh quick charger and battery set b90su). Note the presence of product codes such as "str-de197", "strde197", "b-90su". It is likely that DL methods treat these as out-of-vocabulary words and cannot handle them as well as Sparkly.

We have also examined correct matches found by Sparkly but not found by kNN-jaccard. An interesting pattern is the presence of common tokens. For example, the Hospital Dirty dataset has many names such as "David Smith MD Physician" and "Julie Smith MD Physician". As mentioned earlier, when blocking on these names, tf/idf-based Sparkly can "ignore" the common tokens "MD", "Physician", whereas standard Jaccard and cosine measures cannot.

Cases of Possible Low Recall: We now analyze cases where Sparkly may achieve low recall. Consider a match $g = (u \in B, v \in A)$. There are three possible reasons why tuple v may not make it into the top-k list of u (thus excluding g from the blocking output). First, v may have a low tf/idf score with u. This can happen due to dirty data, missing values, synonyms, and natural variations (e.g., "Robert Smith" vs. "Ben Smith"). Another possibility is that if the data is highly numeric or textual (such as long documents), the tf/idf score may also be low, as we discussed below.

Second, v may have a high tf/idf score with u, but there are more than k true matches for u, so v is excluded.

Finally, many other non-matching tuples in A may have high tf/idf scores with u, crowding out v. For example, if we block just on the person name, then the top-k list for a particular "David Smith" may contain tuples of many other "David Smith"-s, because David Smith is a very common name. As another example, the non-match ("iPhone 12 mint condition 32G white with case", "iPhone 12 mint condition 32G black with case") may have a high tf/idf score, because tf/idf fails to locate the colors and realize that if the colors do not match then the tuples do not match.

To address the above limitations, possible solutions include ways to clean and standardize the data, using a dynamic k value (e.g., if all tf/idf scores in the current top-k list is high, then increase k then query again), and performing information extraction to isolate important attributes such as color. Managing synonyms is also critically important, as synonyms are pervasive in real-world data.

Blocking Numeric and Document Datasets: We have just discussed the possibility that when the data is highly numeric or textual (e.g., long documents), the tf/idf score may be low and so Sparkly may not be as effective. We now examine these possibilities.

We first consider numeric data. It is very difficult to find numeric public datasets with complete gold. After an extensive search, we settle on two datasets AW and RE. Each dataset matches the columns of the tables within a data lake. As such, each dataset consist of a single table X where each tuple describes a column (listing column name, table name, the average length of the column's values, the average/min/max of the column's values if it is numeric, etc.). The goal then is to match X with itself. AW (AdventureWorks) and RE (Real Estate) have 799 and 451 tuples, respectively.

Here we found that SM is better than SA. On AW and RE, SM achieves 81% and 94% recall at $k = 50$ compared to 79% and 67% for SA. This is because SA was confused by numeric attributes and so picked up some numeric attributes to block on. SM is comparable to kNN-jaccard and kNN-cosine, and is much better than the two DL methods and JedAI.

SM however can still be improved. For example, a separate work on schema matching for data lakes (under preparation, from where we obtained the above two datasets) extended SM to incorporate rules such as "if the average values of two numeric columns are too far apart, e.g., one value is greater than 10 times the other value, then do not include the columns as a pair in the blocking output". This solution achieves recall 99% and 97% at $k = 50$ for AW and RE. It turns out that we can naturally incorporate many such rules into the querying function of Lucene.

Thus, the results suggest that *Sparkly still performs better or comparable to SOTA methods on numeric data. Further, it can significantly be improved with rules exploiting the properties of numeric attributes, and many such rules can naturally be incorporated into Lucene.* Future work should explore this direction, and also improve SA to avoid picking up numeric attributes to block on.

We now consider document data. A prior work [27] has created Companies, a document dataset, which has two tables A and B, where each tuple is a long document describing a company (e.g., a document was obtained by crawling the company's Website, then processing to remove all HTML tags).

On Companies, SM obtained recall of 63% at $k = 50$. In contrast, Autoencoder and Hybrid obtain recall of 55 and 43%. kNN-jacc and kNN-cosine obtain recall of 30 and 56%. Thus, *on this long document dataset, Sparkly still outperforms existing SOTA methods.* (Note that here the whole tuple has just one attribute, whose values are long documents. So we just block on this attribute, and there is no reason to use SA.)

But it turns out that we can still improve Sparkly. Recall that SM+ obtained recall of 62%. In contrast, TFIDF-cosince obtained recall of 74%. All of these methods use the default 3gram tokenizer.

Upon closer inspection, we realize that using the 3gram tokenizer is not ideal for long textual documents. So we switched to a world-level tokenizer. This improves the recall significantly at $k = 50$. The recall for SM, SM+, and TFIDF-cosine is now 84%, 87%, and 89%. Note that SM+ is better than SM, and interesting TFIDF-cosine is still a bit better than SM+, but the difference is small. Thus, *Sparkly can still be improved, e.g., by using a tokenizer better suited for long documents, and SM+ is still competitive (very close to the best method). But there is still room for improvement, to reach recall of high 90s.*

Updates: We now discuss how Sparkly can handle updates. A very appealing property of BM25 is that it enables very fast incremental index updates, when we add or remove tuples from the indexed table. In contrast, TFIDF-cosine requires the entire index to be rebuilt from scratch (see the Appendix for a discussion).

Fast index update can provide moderate to significant benefits in many cases. As a case of moderate benefits, consider blocking two tables A and B. Suppose Sparkly has built an index I on A and used it to perform blocking. Now suppose we add a new tuple t to B. Then we do not have to update I. We can just perform blocking between t and A, by using t to probe I. The case of removing a tuple t from B can also be easily handled.

Now suppose we add a tuple t to Table A. We can quickly update index I into I', which incorporates t. However, we need to re-process all tuples in B, since for each such tuple, its top-k list may have changed. So our saving is only in the index construction step.

As a case of significant benefits, consider a table X that contains all customers of a company (typically called Customer 360 in practice). Users often must perform real-time matching of a new tuple t into table X (e.g., to find if a particular person is already in X), and this matching is often done by a blocking step followed by a matching step. Table X is frequently updated (e.g., adding/removing customers). It is critical that such updates are processed quickly, so that real-time matching is always done on the latest version of X. Sparkly is well suited for such cases, because we can update the inverted index I on Table X very quickly.

Scalability, Predictability, and Extensibility: We believe Sparkly scales due to four reasons:

- First, it *decomposes blocking into executing a large number of independent tasks* (each querying the inverted index I using a chunk of tuples in table B).
- Second, it executes these tasks on a Spark cluster in *a distributed share-nothing fashion*, by shipping the index I to the Spark worker nodes, then execute the tasks there.
- Third, it can execute each task fast, by *using the block-max WAND indexing technique* of Lucene.
- Finally, when the table A is too big (e.g., 200M tuples), it breaks A into smaller partitions (e.g., of 50M tuples each) and blocks each partition against table B (this minimizes the problem of having the indexes and other intermediate data structures grow uncontrolled as the table sizes increase, eventually crashing the cluster).

As such, *Sparkly can maximally utilize the entire Spark cluster, and scale horizontally by adding more nodes.* In practice, we have successfully used Sparkly to perform blocking for tables of up to 180M tuples, using Spark clusters of up to 100 nodes.

The above architecture is also *predictable*. Recall that we chop table B into chunks of tuples, and execute these chunks (i.e., use them to query index I) on the worker nodes. After executing a few hundred chunks, it is possible to use the execution times of these chunks to estimate with high accuracy how much longer Sparkly will run (e.g., using Equation 3). We have developed such an estimator, but will not report here for space reasons.

Finally, the above architecture is also *extensible*, in that we can add other kinds of blocking. For example, consider hash blocking. We can create a hash H of table A and ship it to the worker nodes. Then given a tuple $b \in B$, we can consult the inverted index I to obtain a top-k result, consult the hash H to obtain a result, union or intersect the two results, then send the output back to the driver node. In general, we can ship all kinds of indexes to each worker node, then do processing for each tuple $b \in B$ using these indexes.

Thus, we believe that *future blocking solutions should seriously consider an architecture similar to Sparkly, which can provide significant benefits in scaling, predictability, and extensibility.*

Future Research Directions: The current work has shown that (a) tf/idf is highly promising for blocking, and (b) we can develop a tf/idf-based system, Sparkly, that is already practical for large datasets. But a lot more remain to be done.

Based on the discussion so far, we propose to consider the following research directions:

- We should study tf/idf blocking in more depth, improving Sparkly and similar tf/idf systems. The issue of which scoring functions (including tokenizers) are the best for which cases requires more investigation.
- It is important to develop a much bigger benchmark (which has a lot more datasets of diverse characteristics) for blocking. The current datasets are too small (or some are large but have no gold matches, so we cannot evaluate blocking recall), and do not contain enough variety.
- We should develop effective methods to clean and standardize datasets, as this can significantly improve blocking recall and minimize the need to use sophisticated but costly blocking methods.
- We should study how to improve existing blocking solutions and develop new blocking solutions, using Sparkly as a benchmark.
- It is likely that an ideal blocking solution will have to use multiple blocking techniques, including tf/idf and others, to maximize recall. For example, if a company has developed a great blocking method specifically designed to block person names, then when applied to person-name datasets, they may want to use this blocking method, possibly augmented with tf/idf blocking. And when applied to other kinds of datasets, they may just want to use tf/idf blocking. An ideal blocking solution should allow such flexibility.
  The solution must also scale, i.e., block tables of hundreds of millions of tuples in a reasonable time on a reasonable hardware at a reasonable cost. Toward this goal, the scalable share-nothing architecture of Sparkly provides a promising starting point.
- Related to the last point, a lot of existing blocking work has focused on algorithmic development, rather than system aspects. We should devote more effort to system aspects, such as examining desirable properties for a good blocking system, develop open-source systems with these properties, then deploy, evaluate, and improve them. Example questions include "do we need a spectrum of blocking systems, or just one system?", "if we need multiple systems, how do they relate to each other and where to use what?", "what should be the architecture of such systems? the role of a share-nothing architecture? the role of indexes?", etc.

6 ADDITIONAL RELATED WORK

We have discussed related work throughout this paper. We now discuss additional related work. EM has been a long-standing challenge in data management (see [2, 7, 8, 13, 15, 28, 31] for recent books and surveys). There has been multiple academic efforts on building scalable EM systems such as JedAI [32, 34], Magellan [20], and CloudMatcher [18].

Over the past decades, numerous blocking solutions have been developed. See [6, 15, 30, 33] for surveys, and see Section 2 for a discussion of the main blocker categories. However, tf/idf blocking has received very little attention, as far as we can tell. The only work we have found (and cited by the above surveys) is the work [26]. Consider deduplicating a table A, i.e., matching A with A. This work proposes a tf/idf blocking solution that works as follows:

(1) Take a random tuple $x \in A$ then finds the top-k tuples in A with the highest tf/idf scores, using an inverted index on A. Pair x with these tuples and output the pairs.
(2) For tuples that are very close to x, where "close" means (a) within the top-m of x (here $m < k$ and both are pre-specified) or (b) the similarity score exceeds a threshold, remove these tuples from A. Also remove tuple x from A.
(3) Repeat Steps 1-2 until A is exhausted.

As described, this solution works on a single machine and is hard to scale, especially in a share-nothing fashion. A later work [6] experimentally shows that it does not perform better (in recall, output size, and runtime) than other blocking solutions.

Since then we are not aware of any other work that examines tf/idf blocking. The closest work that we have found is the recent work [29], which performs token blocking, i.e., hashing each tuple to multiple blocks, each corresponding to a token in the tuple. This work removes blocks that correspond to tokens of low tf/idf values. The work [5] develops a scoring function that can be viewed as the Jaccard similarity function using IDF. We evaluated this function in Section 5.

The tf/idf measure has long been used in IR and Web search [25]. Lucene, which performs tf/idf search, was released in 1999. For a long time it was somewhat slow and inaccurate, and was largely ignored by the academia [19]. In 2015, however, Lucene adopted cutting-edge techniques such as BM25 and block-max WAND. It is now viewed as quite accurate and fast, and has attracted attention from IR researchers [19]. TF/IDF has long been used in the matching step of EM [9].

Given a set of strings D, a similarity score s, and a query string q, *threshold-based string similarity search* is the problem of finding all strings $d \in D$ such that $s(d, q) \geq t$, where t is a pre-specified threshold. *Top-k string similarity search* is the problem of finding the top-k strings $d \in D$ with the highest similarity score with q. Both problems have been extensively studied [43]. Top-k string similarity search is clearly most related to Sparkly. However, most works have considered only similarity measures such as overlap, Jaccard, cosine, dice, and edit distance. As far as we can tell, top-k tf/idf search has been studied intensively by IR researchers (resulting in the block-max WAND technique), but not by database researchers.

The work [44] develops AutoBlock, a blocking solution which uses labeled data to find a good blocker. Sparkly does not require labeled data (i.e., correct matches). The work [40] shows that AutoBlock was outperformed by the DL methods Autoencoder and Hybrid, which in turn are shown in this work to be outperformed by Sparkly.

The work [23] also addresses blocking. But it maximizes recall while keeping precision (i.e., the fraction of pairs in the blocking output that are correct matches) above a threshold. We consider a fundamentally different problem of maximizing recall for any given k (i.e., any given blocking output size). As such, it is not yet clear how to adapt their techniques to our context. In particular, that work requires estimating the precision of the blocking configs in their setting, which is not possible for the blocking config space that we consider, because our configs do not specify a k value.

Given two sets of strings D and E and a similarity score s, *threshold-based string similarity join* finds all pairs $(d \in D, e \in E)$,

where $s(d, e) \geq t$, with t being a pre-specified threshold. Similarly, *top-k string similarity join* finds the top-k pairs (d, e) with the highest similarity score. These two problems have also been extensively studied (e.g., [42, 43]), but only for overlap, Jaccard, cosine, dice, and edit distance, as far as we can tell.

The share-nothing architecture of Sparkly is reminiscent of share-nothing architectures that have traditionally been studied for parallel processing of relational data [39], and our Spark-based probing method for blocking is reminiscent of distributed/parallel joins for relational data [21, 22]. But here we consider the novel context of blocking for EM. Finally, the work [3] describes an industrial blocking solution at Amazon, which uses meta blocking to manage token-centric blocks and uses sophisticated techniques to scale.

7 CONCLUSIONS

Despite decades of research, tf/idf blocking has received very little attention. Yet anecdotal evidence suggests that it can do very well. As a result, in this paper we have performed an in-depth examination of tf/idf blocking.

We developed Sparkly, a novel solution that performs top-k tf/idf blocking, using Lucene and Spark in a distributed share-nothing architecture. We developed techniques to select good attribute/tokenizer pairs to block on, making Sparkly completely automatic. Extensive experiments show that Sparkly outperforms 8 state-of-the-art blocking solutions and scales to large datasets. We analyzed reasons for Sparkly's strong performance and identified promising future research directions.

Overall, our work suggests that tf/idf blocking should receive more attention, that future blocking work should consider Sparkly as a baseline, and that the distributed share-nothing architecture of Sparkly provides a promising starting point to build blocking solutions that are scalable, predictable, and extensible.

Acknowledgments: We are grateful to the reviewers for the insightful comments that greatly improve this paper. We thank George Papadakis and Themis Palpanas for their assistance with understanding and running experiments with JedAI, and Saravanan Thirumuruganathan for his assistance with the DeepBlocker software.

REFERENCES

[1] [n.d.]. Benchmark datasets for entity resolution. https://dbs.uni-leipzig.de/en/research/projects/object_matching/fever/benchmark_datasets_for_entity_resolution.

[2] Nils Barlaug and Jon Atle Gulla. 2020. Neural networks for entity matching. *arXiv preprint arXiv:2010.11075* (2020).

[3] Andrew Borthwick, Stephen Ash, Bin Pang, Shehzad Qureshi, and Timothy Jones. 2020. Scalable Blocking for Very Large Databases. In *ECML PKDD 2020 Workshops - Workshops of the European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD 2020): SoGood 2020, PDFL 2020, MLCS 2020, NFMCP 2020, DINA 2020, EDML 2020, XKDD 2020 and INRA 2020, Ghent, Belgium, September 14-18, 2020, Proceedings (Communications in Computer and Information Science)*, Irena Koprinska, Michael Kamp, Annalisa Appice, Corrado Loglisci, Luiza Antonie, Albrecht Zimmermann, Riccardo Guidotti, Özlem Özgöbek, Rita P. Ribeiro, Ricard Gavaldà, João Gama, Linara Adilova, Yamuna Krishnamurthy, Pedro M. Ferreira, Donato Malerba, Ibéria Medeiros, Michelangelo Ceci, Giuseppe Manco, Elio Masciari, Zbigniew W. Ras, Peter Christen, Eirini Ntoutsi, Erich Schubert, Arthur Zimek, Anna Monreale, Przemyslaw Biecek, Salvatore Rinzivillo, Benjamin Kille, Andreas Lommatzsch, and Jon Atle Gulla (Eds.), Vol. 1323. Springer, 303–319. https://doi.org/10.1007/978-3-030-65965-3_20

[4] Andrei Z. Broder, Michael Herscovici, and Jason Zien. 2003. Efficient query evaluation using a two-level retrieval process. In *In Proc. of the 12th ACM Conf. on Information and Knowledge Management.*

[5] Surajit Chaudhuri, Kris Ganjam, Venkatesh Ganti, and Rajeev Motwani. 2003. Robust and Efficient Fuzzy Match for Online Data Cleaning. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, Alon Y. Halevy, Zachary G. Ives, and AnHai Doan (Eds.). ACM, 313–324. https://doi.org/10.1145/872757.872796

[6] Peter Christen. 2011. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE transactions on knowledge and data engineering* 24, 9 (2011), 1537–1555.

[7] Peter Christen. 2012. *Data Matching - Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection.* Springer. https://doi.org/10.1007/978-3-642-31164-2

[8] Vassilis Christophides, Vasilis Efthymiou, Themis Palpanas, George Papadakis, and Kostas Stefanidis. 2020. An overview of end-to-end entity resolution for big data. *ACM Computing Surveys (CSUR)* 53, 6 (2020), 1–42.

[9] William W. Cohen, Pradeep Ravikumar, and Stephen E. Fienberg. 2003. A Comparison of String Distance Metrics for Name-Matching Tasks. In *Proceedings of IJCAI-03 Workshop on Information Integration on the Web (IIWeb-03), August 9-10, 2003, Acapulco, Mexico*, Subbarao Kambhampati and Craig A. Knoblock (Eds.). 73–78. http://www.isi.edu/info-agents/workshops/ijcai03/papers/Cohen-p.pdf

[10] S. Das et al. 2017. Falcon: scaling up hands-off crowdsourced entity matching to build cloud services. SIGMOD.

[11] Constantinos Dimopoulos, Sergey Nepomnyachiy, and Torsten Suel. 2013. Optimizing top-k document retrieval strategies for block-max indexes. In *Sixth ACM International Conference on Web Search and Data Mining, WSDM 2013, Rome, Italy, February 4-8, 2013*, Stefano Leonardi, Alessandro Panconesi, Paolo Ferragina, and Aristides Gionis (Eds.). ACM, 113–122. https://doi.org/10.1145/2433396.2433412

[12] Shuai Ding and Torsten Suel. 2011. Faster top-k document retrieval using block-max indexes. In *Proceeding of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2011, Beijing, China, July 25-29, 2011*, Wei-Ying Ma, Jian-Yun Nie, Ricardo Baeza-Yates, Tat-Seng Chua, and W. Bruce Croft (Eds.). ACM, 993–1002. https://doi.org/10.1145/2009916.2010048

[13] A. Doan, A. Halevy, and Z. Ives. 2012. *Principles of Data Integration.* Elsevier.

[14] Muhammad Ebraheem, Saravanan Thirumuruganathan, Shafiq Joty, Mourad Ouzzani, and Nan Tang. 2018. Distributed representations of tuples for entity resolution. *PVLDB* 11, 11 (2018), 1454–1467.

[15] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. 2007. Duplicate Record Detection: A Survey. *TKDE* 19, 1 (2007).

[16] 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.

[17] Yash Govind, Pradap Konda, Paul Suganthan G. C., Philip Martinkus, Palaniappan Nagarajan, Han Li, Aravind Soundararajan, Sidharth Mudgal, Jeffrey R. Ballard, Haojun Zhang, Adel Ardalan, Sanjib Das, Derek Paulsen, Amanpreet Singh Saini, Erik Paulson, Youngchoon Park, Marshall Carter, Mingju Sun, Glenn Moo Fung, and AnHai Doan. 2019. Entity Matching Meets Data Science: A Progress Report from the Magellan Project. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 389–403. https://doi.org/10.1145/3299869.3314042

[18] Yash Govind, Erik Paulson, Palaniappan Nagarajan, Paul Suganthan G. C., AnHai Doan, Youngchoon Park, Glenn Fung, Devin Conathan, Marshall Carter, and Mingju Sun. 2018. CloudMatcher: A Hands-Off Cloud/Crowd Service for Entity Matching. *Proc. VLDB Endow.* 11, 12 (2018), 2042–2045. https://doi.org/10.14778/3229863.3236255

[19] Adrien Grand, Robert Muir, Jim Ferenczi, and Jimmy Lin. 2020. From MAXSCORE to Block-Max Wand: The Story of How Lucene Significantly Improved Query Evaluation Performance. In *Advances in Information Retrieval - 42nd European Conference on IR Research, ECIR 2020, Lisbon, Portugal, April 14-17, 2020, Proceedings, Part II (Lecture Notes in Computer Science)*, Joemon M. Jose, Emine Yilmaz, João Magalhães, Pablo Castells, Nicola Ferro, Mário J. Silva, and Flávio Martins (Eds.), Vol. 12036. Springer, 20–27. https://doi.org/10.1007/978-3-030-45442-5_3

[20] Pradap Konda, Sanjib Das, Paul Suganthan GC, AnHai Doan, Adel Ardalan, Jeffrey R Ballard, Han Li, Fatemah Panahi, Haojun Zhang, Jeff Naughton, et al. 2016. Magellan: Toward building entity matching management systems. *PVLDB* 9, 13 (2016), 1581–1584.

[21] Donald Kossmann. 2000. The State of the art in distributed query processing. *ACM Comput. Surv.* 32, 4 (2000), 422–469. https://doi.org/10.1145/371578.371598

[22] Paraschos Koutris, Semih Salihoglu, and Dan Suciu. 2018. Algorithmic Aspects of Parallel Data Processing. *Found. Trends Databases* 8, 4 (2018), 239–370. https://doi.org/10.1561/1900000055

[23] Peng Li, Xiang Cheng, Xu Chu, Yeye He, and Surajit Chaudhuri. 2021. Auto-FuzzyJoin: Auto-Program Fuzzy Similarity Joins Without Labeled Examples. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1064–1076. https://doi.org/10.1145/3448016.3452824

[24] Yuliang Li, Jinfeng Li, Yoshihiko Suhara, AnHai Doan, and Wang-Chiew Tan. 2020. Deep entity matching with pre-trained language models. *PVLDB* 14, 1 (2020), 50–60.

[25] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to information retrieval.* Cambridge University Press. https://doi.org/10.1017/CBO9780511809071

[26] Andrew McCallum, Kamal Nigam, and Lyle H. Ungar. 2000. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining, Boston, MA, USA, August 20-23, 2000*, Raghu Ramakrishnan, Salvatore J. Stolfo, Roberto J. Bayardo, and Ismail Parsa (Eds.). ACM, 169–178. https://doi.org/10.1145/347090.347123

[27] Sidharth Mudgal, Han Li, Theodoros Rekatsinas, AnHai Doan, Youngchoon Park, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, and Vijay Raghavendra. 2018. Deep learning for entity matching: A design space exploration. In *SIGMOD*.

[28] Felix Naumann and Melanie Herschel. 2010. An introduction to duplicate detection. *Synthesis Lectures on Data Management* 2, 1 (2010), 1–87.

[29] Kevin O'Hare, Anna Jurek-Loughrey, and Cassio P. de Campos. 2022. High-Value Token-Blocking: Efficient Blocking Method for Record Linkage. *ACM Trans. Knowl. Discov. Data* 16, 2 (2022), 24:1–24:17. https://doi.org/10.1145/3450527

[30] Kevin OâĂŹHare, Anna Jurek-Loughrey, and Cassio de Campos. 2019. A review of unsupervised and semi-supervised blocking methods for record linkage. *Linking and Mining Heterogeneous and Multi-view Data* (2019), 79–105.

[31] George Papadakis, Ekaterini Ioannou, Emanouil Thanos, and Themis Palpanas. 2021. The Four Generations of Entity Resolution. *Synthesis Lectures on Data Management* 16, 2 (2021), 1–170.

[32] George Papadakis, George Mandilaras, Luca Gagliardelli, Giovanni Simonini, Emmanouil Thanos, George Giannakopoulos, Sonia Bergamaschi, Themis Palpanas, and Manolis Koubarakis. 2020. Three-dimensional Entity Resolution with JedAI. *Information Systems* 93 (2020), 101565.

[33] George Papadakis, Dimitrios Skoutas, Emmanouil Thanos, and Themis Palpanas. 2020. Blocking and filtering techniques for entity resolution: A survey. *ACM Computing Surveys (CSUR)* 53, 2 (2020), 1–42.

[34] George Papadakis, Leonidas Tsekouras, Emmanouil Thanos, Nikiforos Pittaras, Giovanni Simonini, Dimitrios Skoutas, Paul Isaris, George Giannakopoulos, Themis Palpanas, and Manolis Koubarakis. 2020. JedAI3: beyond batch, blocking-based Entity Resolution.. In *EDBT*. 603–606.

[35] D. Paulsen, Y. Govind, and A. Doan. 2022. *Homepage of the Sparkly Blocking System.* Technical Report. http://pages.cs.wisc.edu/~anhai/sparkly.html.

[36] G. Ppadakis, M. Fisichella, F. Schoger, G. Mandilaras, N. Augsten, and W. Nejdl. 2022. *Benchmarking Filtering Techniques for Entity Resolution.* Technical Report. arXiv:2022.12521v3.

[37] Anna Primpeli, Ralph Peeters, and Christian Bizer. 2019. The WDC Training Dataset and Gold Standard for Large-Scale Product Matching. In *Companion of The 2019 World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, Sihem Amer-Yahia, Mohammad Mahdian, Ashish Goel, Geert-Jan Houben, Kristina Lerman, Julian J. McAuley, Ricardo Baeza-Yates, and Leila Zia (Eds.). ACM, 381–386. https://doi.org/10.1145/3308560.3316609

[38] Rudi Seitz. 2022. *Understanding tf/idf and BM25.* Technical Report. https://kmwllc.com/index.php/2020/03/20/understanding-tf-idf-and-bm-25.

[39] Michael Stonebraker. 1986. The Case for Shared Nothing. *IEEE Database Eng. Bull.* 9, 1 (1986), 4–9. http://sites.computer.org/debull/86MAR-CD.pdf

[40] Saravanan Thirumuruganathan, Han Li, Nan Tang, Mourad Ouzzani, Yash Govind, Derek Paulsen, Glenn Fung, and AnHai Doan. 2021. Deep Learning for Blocking in Entity Matching: A Design Space Exploration. *Proc. VLDB Endow.* 14, 11 (2021), 2459–2472. https://doi.org/10.14778/3476249.3476294

[41] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), 80–83. http://www.jstor.org/stable/3001968

[42] C. Xiao, W. Wang, X. Lin, and H. Shang. 2009. Top-k set similarity joins. ICDE.

[43] Minghe Yu, Guoliang Li, Dong Deng, and Jianhua Feng. 2016. String similarity search and join: a survey. *Frontiers Comput. Sci.* 10, 3 (2016), 399–417. https://doi.org/10.1007/s11704-015-5900-5

[44] Wei Zhang, Hao Wei, Bunyamin Sisman, Xin Luna Dong, Christos Faloutsos, and Davd Page. 2020. AutoBlock: A hands-off blocking framework for entity matching. In *WSDM*. 744–752.

A BLOCKING FOR DEDUPLICATION AND TABLES WITH DIFFERENT SCHEMAS

Deduplication: Deduplication means finding tuple pairs (a_i, a_j) within the same table A that match. We can also use Sparkly to do blocking for such cases, by pretending that we are matching table A with a copy of itself. That is, we will index A and then use it for search as well. The blocking output then likely contains duplicate pairs, e.g., (a_i, a_j) and (a_j, a_i). These duplicate pairs can then be removed in multiple ways, the simplest being removing them using built-in Spark operations.

Tables with Different Schemas: Given two tables A and B with different schemas, we ask the user to manually identify matching attributes between the two schemas, e.g., location = address, lastname = surname, concat(city,state) = address. We use these matches to transform the two tables into two new tables A' and B' with the same schema, then apply Sparkly.

B EXPLAINING THE AUC FORMULA

We now explain how we compute $AUC(b, L, k)$, the discriminativeness of config L for a tuple $b \in B$, as the normalized AUC. Let $r(b, L, k) = ((v_1, s_1), ...(v_{k'}, s_{k'}))$ be the top-k tuple list retrieved from index I, for record $b \in B$, scored according to config L, sorted in decreasing order of score $s_1, \ldots, s_{k'}$ ($k' \leq k$ because only tuples with positive score can be in the list). Then we can compute the area under the curve as

$$AUC(b, L, k) = \frac{1}{k' \cdot s_1} \sum_{i=1}^{k'-1} \left(s_{i+1} + \frac{s_i - s_{i+1}}{2} \right).$$

To understand the above formula, imagine we have plotted the curve of the scores $s_1, ...s_{k'}$, where the score for record i is plotted at point (i, s_i), giving us a curve with discrete segments. To take the area under the curve, we can divide the area into rectangles and right triangles, one rectangle and one triangle per segment. Consider the area under the curve of the first segment for $x \in [1, 2]$. The rectangle in this interval is width 1 and height s_2 meaning the area is s_2, corresponding to s_{i+1} in the formula above. For the triangle, one leg is length $s_1 - s_2$ and the other is length 1, hence the area is $\frac{s_1 - s_2}{2}$ corresponding to $\frac{s_i - s_{i+1}}{2}$ in the formula above. We can then repeat this computation for each segment of the curve, giving us the sum over 1 to $k' - 1$. Finally, we normalize the area by dividing by the number of tuples, k', times the max score, s_1, corresponding to the lead coefficient $\frac{1}{k' \cdot s_1}$.

C THE SIX ADDITIONAL DATASETS FOR OUR EXPERIMENTS

We now describe the six additional datasets used in our experiments. BC is Big Citations, which consist of two tables of 2.5M and 1.8M tuples, respectively. Here each tuple is a paper citation. BC has been used in the paper [10] and is available in the Magellan dataset repository.

MB is MusicBrainz, which is a single table of 20M tuples, each describing a song. A copy of MB is available at [1]. MB comes with the complete gold, but this gold is synthetic. WDC is a single table of 26M tuples, each describing a product. WDC is available at [37]. It comes with some gold matches, but not the complete gold.

The two numeric datasets are AW and RE. Each dataset matches the columns of the tables within a data lake. As such, each dataset consist of a single table X where each tuple describes a column (listing column name, table name, the average length of the column's values, the average/min/max of the column's values if it is numeric, etc.). The goal then is to match X with itself. AW (AdventureWorks) and RE (Real Estate) have 799 and 451 tuples, respectively. These datasets come from an ongoing work that matches the schemas of a data lake. They will be made available on Sparkly's homepage shortly.

Companies is a document data set used by a prior work [27]. It has two tables A and B, where each tuple is a long document describing a company (e.g., a document was obtained by crawling the company's Website, then processing to remove all HTML tags). This dataset is available at https://github.com/anhaidgroup/deepmatcher.

D JEDAI BLOCKERS IN OUR EXPERIMENTS

JedAI is a well-known open-source software for EM. In JedAI users can execute a variety of blocking workflows. These workflows proceed in two stages. First, they create a set of initial blocks, by tokenizing each tuple to be matched, creating a block per token, and assigning all tuples containing that token to that block. Then they apply various methods to prune away blocks or drop pairs from the final candidate set, based on various weighting schemes.

JedAI provides many modules for the above stages, such as Cardinality Node Pruning (CNP), Weighted Edge Pruning (WEP), Comparison Propagation, Standard Blocking, Q-Gram Blocking, and Blocking Filtering.

Based on personal communications with the JedAI authors, we compare Sparkly to three state-of-the-art JedAI blocking workflows: JedAI Default Blocking (JD), Parameter-free Blocking Workflow (PBW), and Default Blocking workflow (DBW).

JedAI Default Blocking (JD): JD begins with Standard Blocking, which creates a block for every unique whitespace-delimited token in the dataset. Next, JD applies Comparison-based Block Purging, which removes blocks that generate many pairs (i.e., large blocks). Then it applies Block Filtering, where each entity is removed from blocks which are larger than the median size of the blocks which the entity appears in. For example, if the median block size of the blocks containing record r is 100, then r would be removed from any block that has size greater than 100. Finally, JD applies Cardinality Node Pruning, which retains the top-k pairs for each entity based on a weighting scheme, in this case the Jaccard index of the blocks to which the entities belong.

Parameter Free Blocking Workflow (PBW): PBW begins with Standard Blocking to create a block for every unique whitespace-delimited token in the dataset. Then it applies Comparison-based Block Purging to remove blocks that generate many pairs (i.e., large blocks). Finally, it deduplicates the remaining pairs generated from the blocks, using Comparison Propagation, to generate the final candidate set.

Default Blocking (DBW): DBW begins with 6-Gram Blocking to create a block for every unique 6-gram token in the dataset. Next,

it applies Block Filtering, where each entity is removed from blocks which are larger than the median size of the blocks to which the entity appears in. Finally, DBW applies weighted edge pruning (using the ECBS weighting scheme), which discards pairs that have weight lower than the average for the current candidate set.

E SENSITIVITY ANALYSIS FOR BM25 AND THE CONFIG SEARCHER

BM25's Parameters: BM25 has two parameters: k_1 (default value 1.2) and b (default value 0.75), to handle term saturation and document length. Figures 17-18 show the results of varying these parameters.

Varying k_1 from 1 to 2 does not significantly change SM's performance. This may be because in our setting blocking attributes do not have many terms, and they do not have high term frequency, so term saturation is not a major issue. Varying b from 0.5 to 1 changes SM's performance more, by up to 2% CSSR. But we also find that $b = 0.75$ provides a good default value for SM, as its curve is either the best curve or very close to the best curve on most datasets.

Config Searcher's Parameters: Recall that SA uses a searcher to find a good blocking config. This searcher has four major parameters: (1) the size of B', a sample of table B on which to score the configs (set to 10K), (2) the number of tuples returned in each querying $k = 250$, (3) the number of initial configs selected (set to 10), and (4) the max number of attributes considered in a config (set to 3).

Figure 19 to Figure 22 show the results of varying these parameters. Varying (1) from 5K to 15K changes SA minimally. Varying (2) from 200 to 300 again changes SA minimally (only up to 0.2% CSSR on 1 dataset). Similarly, varying (3) from 8 to 12 and varying (4) from 2 to 4 show minimal changes. In all cases, the default values for (1)-(4) provide a good curve, which is either the best or very near the best.

F SCALABILITY OF VARIOUS BLOCKING METHODS

We now discuss potential problems in scaling SOTA blocking methods considered in this paper.

Deep Learning (DL) Blockers: Recall that we experimented with two SOTA DL blockers: Autoencoder and Hybrid. There are two major issues to contend with when trying to scale DL blockers: model training and top-k computation.

Model Training: The main challenge to scaling training deep learning models is the amount of data that they require to train on. In order to train effectively, the data for deep learning models is typically shuffled each iteration, meaning that the training data is accessed randomly, rather than sequentially. When the training data does not fit in DRAM and is swapped on disk, the random I/O incurred for each training iteration quickly becomes a bottleneck. In order to scale, either more DRAM needs to be added which is costly, or the model must be trained in a distributed set up which greatly increases the complexity of the training procedure.

Training DL models can also take a long time, as we provided numbers in the experiment section.

Top-k Computation: Deep learning based blockers perform top-k search over a set of dense vectors using the cosine of the angles between the vectors as the scoring metric. We are not currently aware of any *exact* algorithms that compute the top-k without resorting to brute force computation (i.e. computing scores for all vectors and sorting). That is, all known methods doing the dense vector search are linear in the number of vectors being searched, meaning that the overall execution time is quadratic. It is possible of course to do approximate top-k search, which is much faster. FAISS is a well-known open-source library for both exact and approximate top-k search.

It is also possible to use GPUs to perform top-k computations. In this case we still do brute-force computations, but we can do a very large number of such computations in parallel on the GPUs, thus speeding up the search. The downside is the need to use powerful GPUs.

Rule-Based Blockers: We experimented with RBB, a SOTA industrial blocker. This blocker asks the user to label a few hundreds of tuple pairs (as match/no-match), then uses the labeled pairs to learn blocking rules. This labeling step takes time, e.g., 152-177 mins in our experiments.

The time to apply the learned blocking rules can also be significant, depending on the size of the tables. For example, applying the blocking rules to Songs tables of size 1M, 3M, 5M takes 7.45, 23, and 326 mins, respectively, in our experiments. (For the table of size 5M, the solution learns a different set of blocking rules and hence incurs longer application time, compared to the case of table of size 3M.)

Token Blockers: Recall that we experimented with 3 JedAI blockers, which belong to the token blocking family. Token blockers can have unpredictable time and memory requirements, because we do not know *a priori* how large each block can be. For example, the PBW blocker required more than 100G of RAM to run the Songs Dirty dataset.

As another example, on the BC (Big Citations) dataset, the DBW workflow crashed (out of memory on a SOTA hardware setting), PBW ran in 92 minutes, and the JedAI default workflow ran in 246 minutes. All three JedAI methods crashed on WDC 10M and MB 10M.

kNN Blockers: We experimented with kNN-Jaccard and kNN-cosine blockers. For kNN-Jaccard, we are not aware of any existing work to efficiently support top-k search for Jaccard (most existing top-k search works have considered only edit distance [43]). As discussed in Section 6, many works have developed solutions for top-k similarity joins, including support for Jaccard [43]. In principle, we can use these solutions to perform top-k search, in a share-nothing fashion on a Spark cluster. But it is unclear how fast such solutions will be.

Similarly, for kNN-cosine, existing work on top-k similarity search has not considered cosine, as far as we can tell. Further, existing work on top-k similarity joins have considered cosine [43] and can be adapted to perform top-k blocking in a share-nothing fashion. But it is unclear how fast such solutions will be. Finally, it is possible to modify the block-max WAND algorithm to evaluate top-k queries for set cosine. At the time of writing, however, we

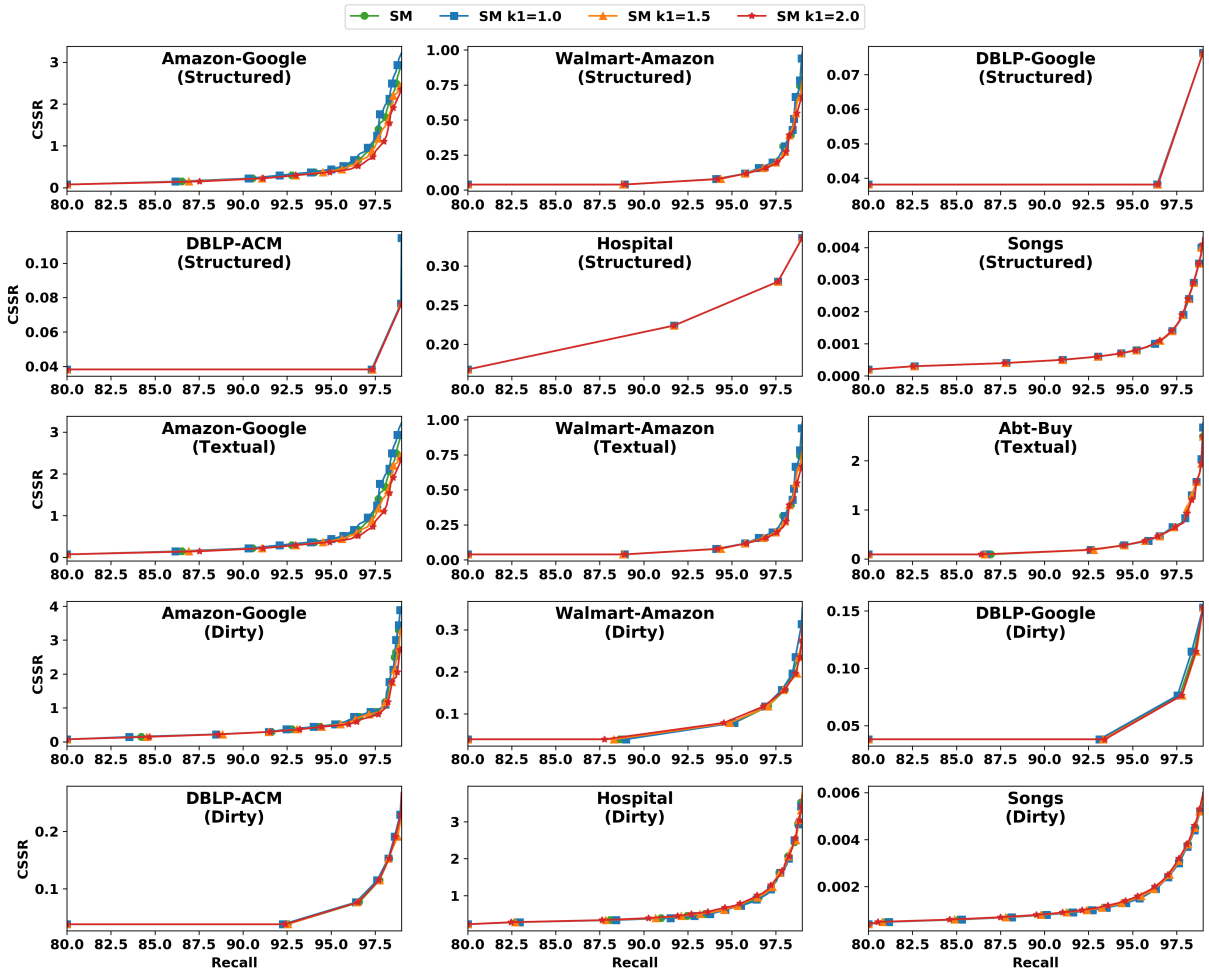

Figure 17: Varying the parameter k_1 of BM25.

are not aware of any system which has implemented the WAND or block-max WAND algorithm for set cosine.

G THE MODIFIED BM25 FORMULA FOR SM+ AND SA+

Recall that the original BM25 scoring function is as follows:

$$s(D,Q) = \sum_{t \in Q} \frac{tf(t,D) \cdot (k_1+1)}{tf(t,D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{avgdl})} \cdot idf(t),$$

where $idf(t) = log(\frac{N - df(t) + 0.5}{df(t) + 0.5} + 1)$, and k_1 and b are free parameters, often set as $k_1 \in [1.2, 2.0]$ and $b = 0.75$.

We modified the above scoring function to incorporate TF and IDF on the query Q's side as follows. We weigh each term in query Q as we would in TFIDF-cosine. That is, for each term t in Q, we multiply by $(\log(tf(t,Q)+1)) * smooth\_idf(t)$, where $smooth\_idf = \log(\frac{N+1}{df(t)+1}) + 1$. This gives us the new scoring function $s(D,Q) = \sum_{t \in Q} \frac{tf(t,D) \cdot (k_1+1)}{tf(t,D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{avgdl})} \cdot idf(t) \cdot ((\log(tf(t))+1) \cdot smooth\_idf(t))$.

H UPDATING THE INDEX

Given the BM25 scoring function

$$s(D,Q) = \sum_{t \in Q} \frac{tf(t,D) \cdot (k_1+1)}{tf(t,D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{avgdl})} \cdot idf(t),$$

it is clear that for the inverted index, we need to keep track of

(1) $idf(t)$ for all t.
(2) the size of each document D.
(3) $avg\ dl$, the average length of documents.
(4) a matrix M that lists documents on one axis and terms on the other axis, and each cell lists $tf(t,D)$ for a term t in document D.

When adding a new document, we need to update $idf(t)$ for all t, since the total number of documents has changed. But doing this, as well as Items 2-3, is quick. Similarly, updating the matrix M in Item 4 is also quick, because we just have to add a new row (or column) for the new document.

In contrast, in TFIDF-cosine, each document D will be converted into a vector and we need to keep tracks of these vectors. We do this using a matrix N, where we list documents on the rows, say,

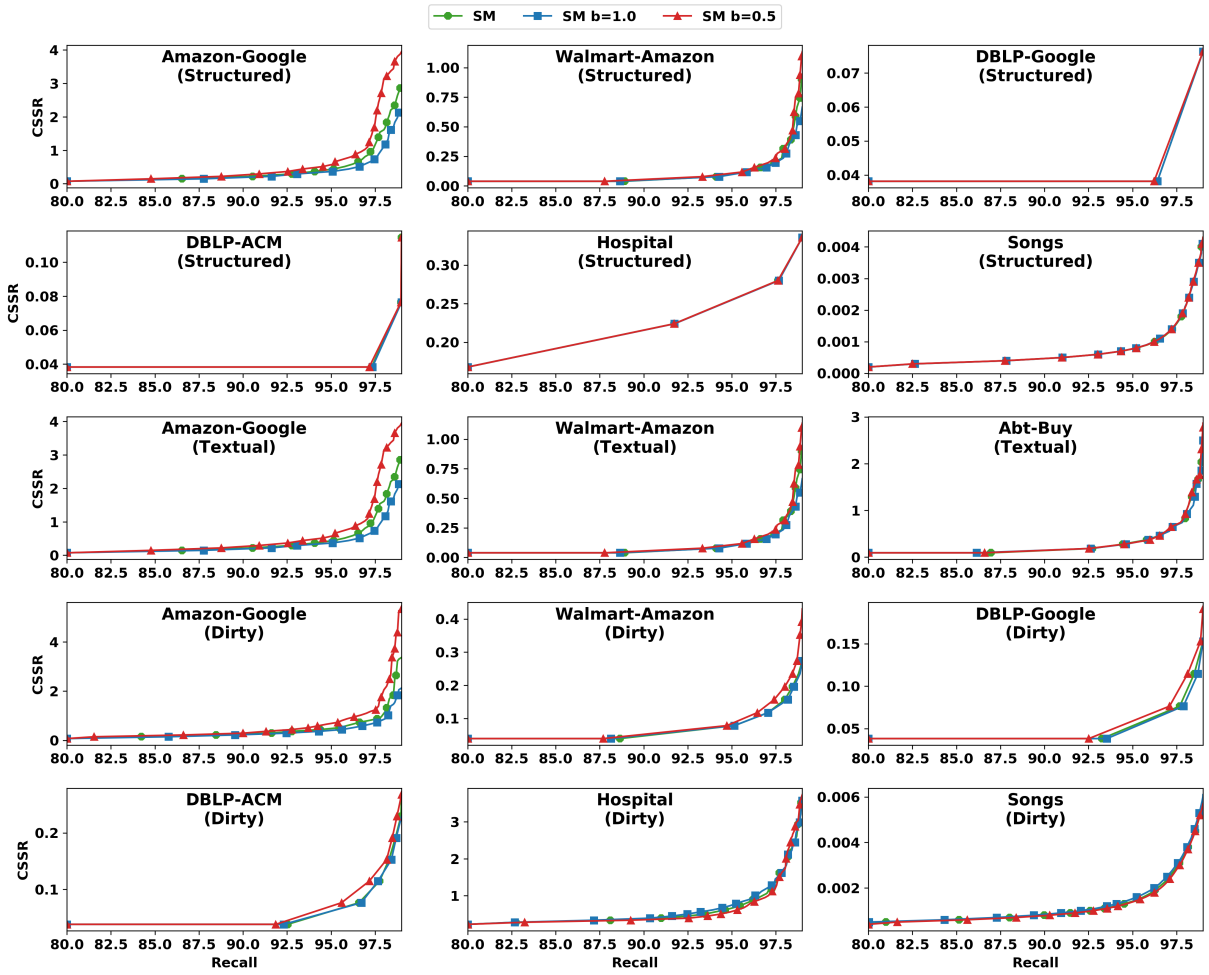

Figure 18: Varying the parameter b of BM25.

and the terms on the columns. Each cell of N lists the value $tf * idf$ normalized by the document length. Now if we add a new document, that will change the IDF of *all* terms (because the total number of documents has increased by 1). This means we also have to update *all* cells in the matrix N. This is a very expensive operation, effectively meaning the entire index must be rebuilt from scratch.

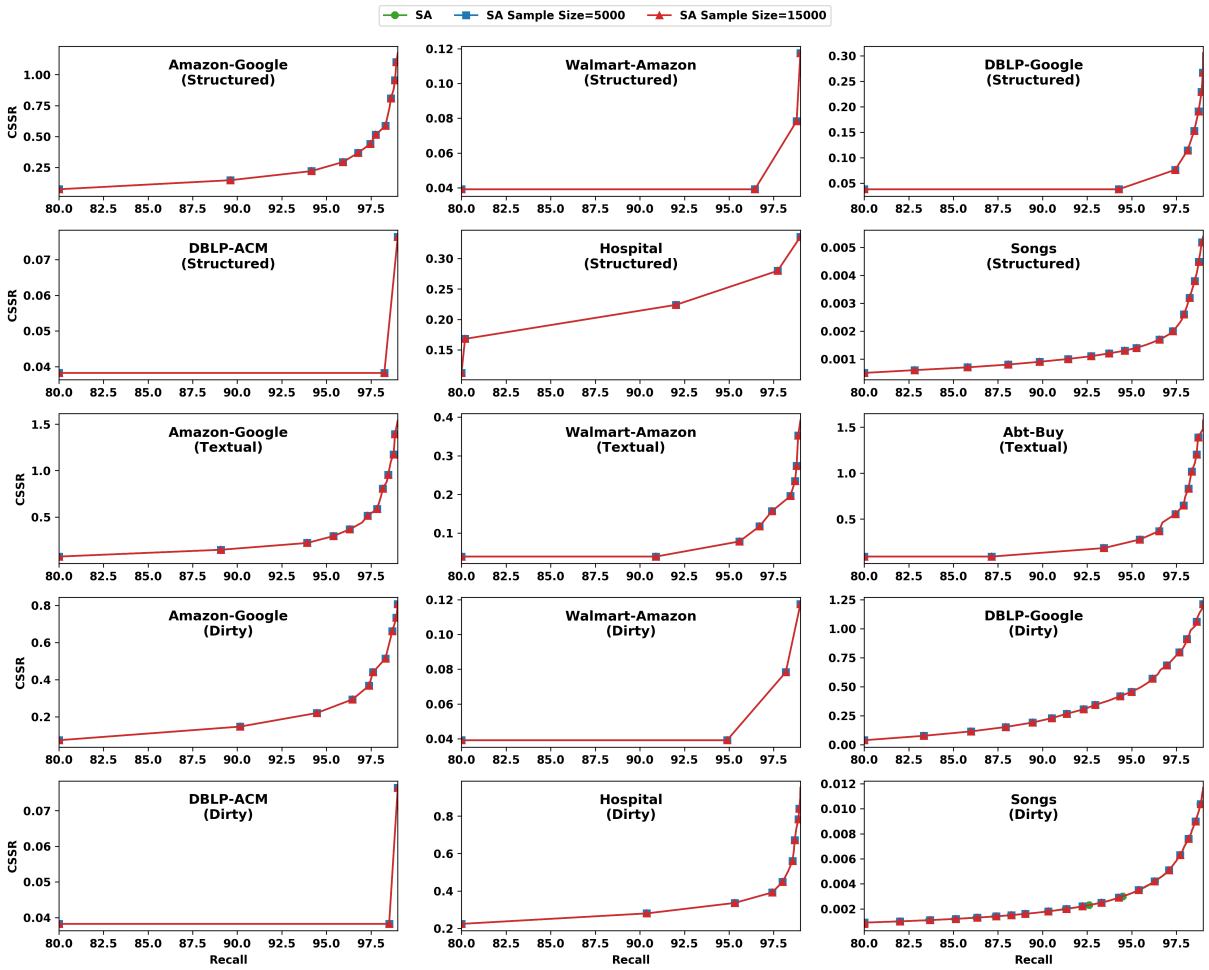

Figure 19: Config searcher: varying the size of B', a sample of table B on which to score the configs; default value is 10K.

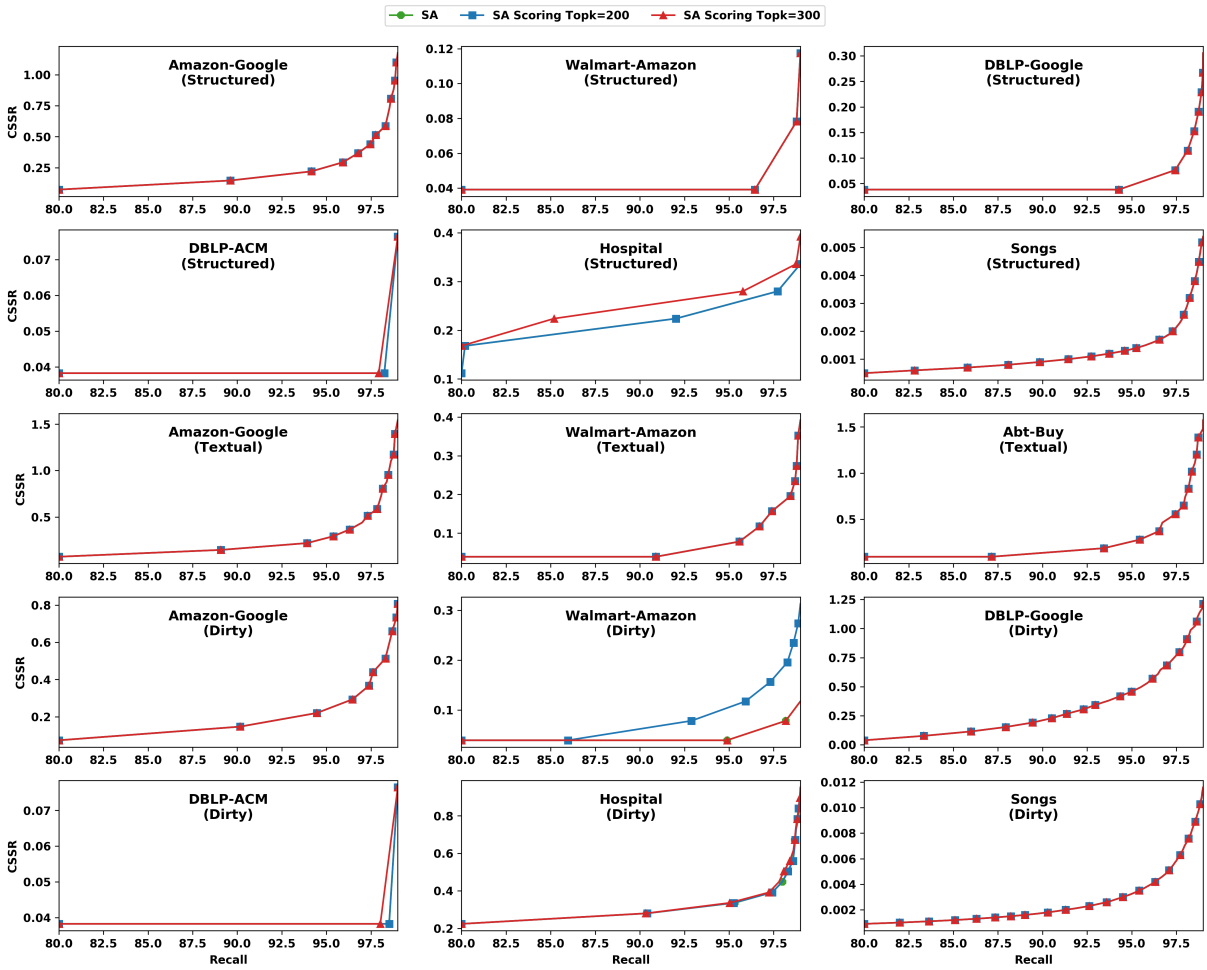

Figure 20: Config searcher: varying the number of tuples returned in each querying; default value is $k = 250$.

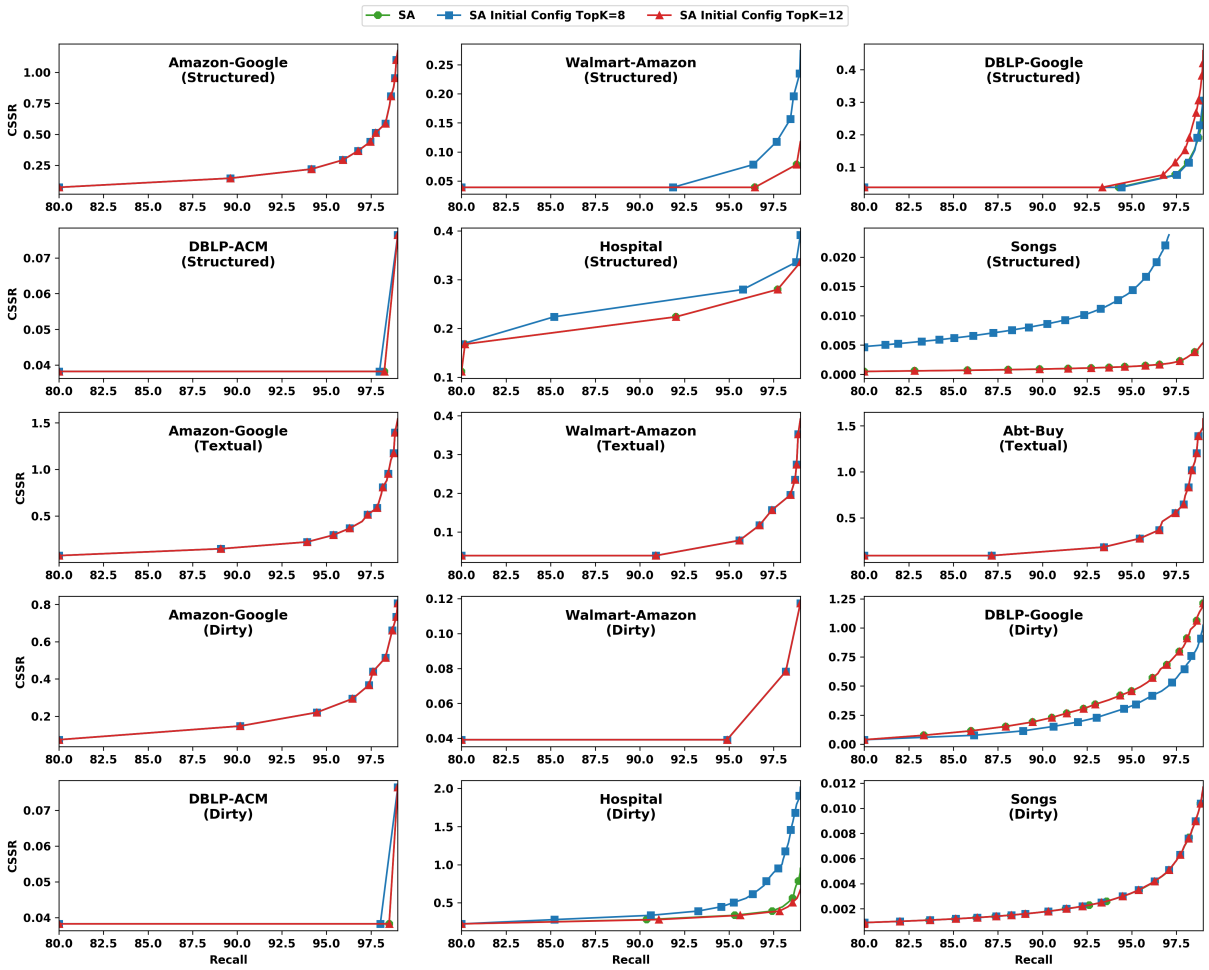

Figure 21: Config searcher: varying the number of initial configs selected for the search; the defaul value is 10.

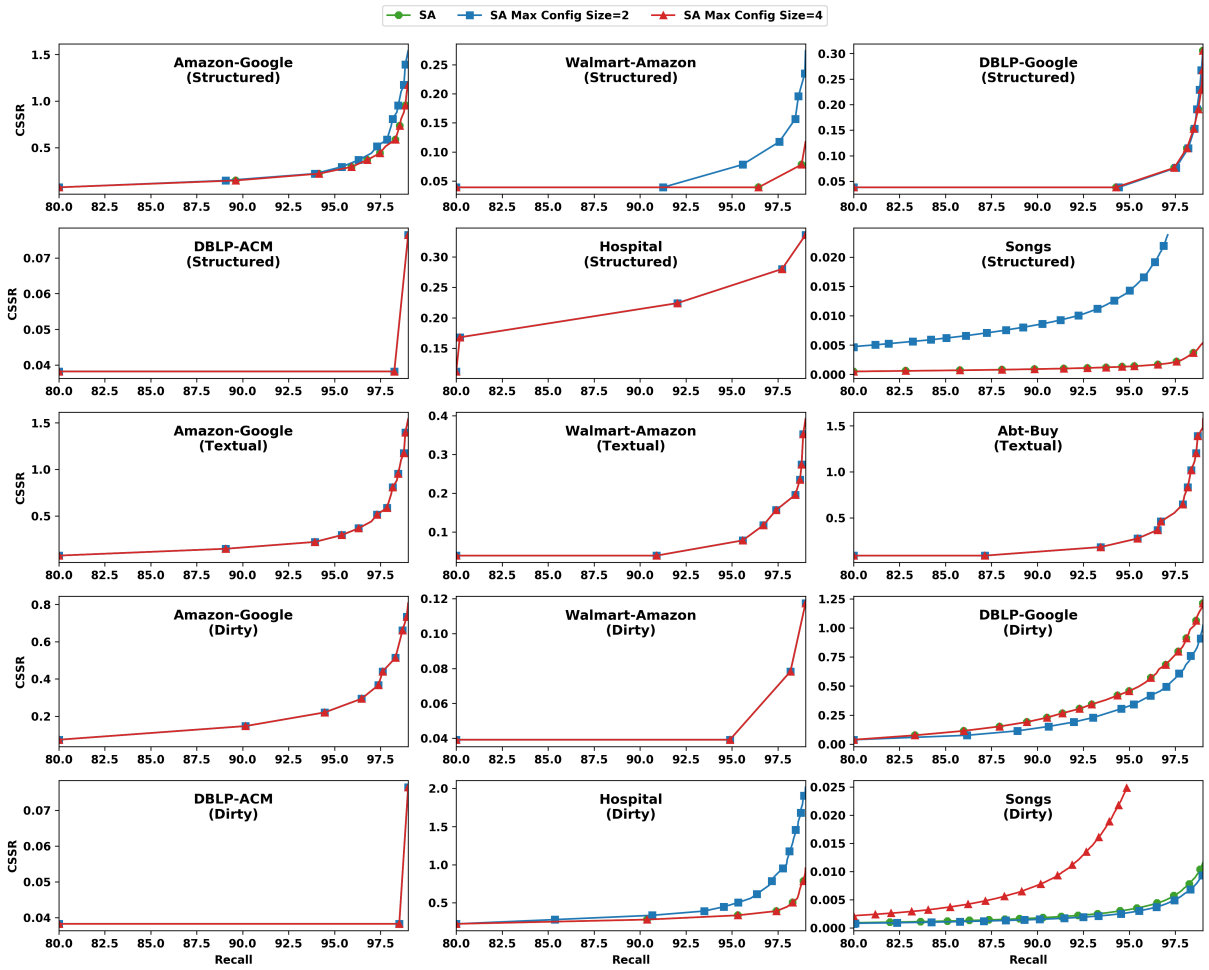

Figure 22: Config searcher: varying the max number of attributes considered in a config; the default value is 3.