TOWARD EFFECTIVE BLOCKING FOR ENTITY MATCHING

By

Derek Paulsen

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2025

Date of final oral examination: 11/21/2025

The dissertation is approved by the following members of the Final Oral Committee:

Mark Craven, Professor, Biostatistics and Medical Informatics, UW-Madison
AnHai Doan, Professor, Computer Sciences, UW-Madison
Goetz Graefe, Affiliated Professor, Computer Sciences, UW-Madison
Paraschos Koutris, Associate Professor, Computer Sciences, UW-Madison

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

In this dissertation we study entity matching, a fundamental problem that lies at the heart of data integration for data science and AI. Specifically, we consider the following common entity matching problem: given two table A and B with the same schema, find all pairs of records $(a, b) \in A \times B$ that "match", i.e., refer to the same real world entity.

Typically, entity matching is done in two steps: blocking and matching. The goal of the blocking step is to quickly reduce the number of pairs to be processed later in the matching step, while retaining as many true matches as possible. The goal of the matching step is to accurately predict which records pairs match. In this dissertation we focus on the blocking step and make three major contributions.

The first contribution is Sparkly, a novel TF-IDF based blocker built on top of Spark and Lucene, using a distributed shared-nothing architecture. The TF-IDF similarity measure is well known in the information retrieval literature but has received very little attention in entity matching research. In developing Sparkly, we explore TF-IDF based blocking for entity matching and demonstrate its effectiveness in a wide range of scenarios. Extensive experiments show that Sparkly outperforms eight state-of-the-art blockers, producing both higher recall and smaller candidate sets. Additionally, we ran Sparkly on over 100M tuples and demonstrate near-linear scale-out behavior.

The second contribution is Delex, a system for combining blocking methods using a powerful declarative language. Delex is built on top of Spark. In real-world applications, users frequently want to combine multiple blocking methods to take advantage of the strengths of each

method. Currently, combining multiple blocking methods is done in an ad-hoc way, which leads to both costly development and suboptimal performance. Delex is designed from the ground up for combining blocking methods using a scalable architecture. Experiments show that Delex can effectively optimize blocking plans, reducing runtime by up to threefold, and can scale to large datasets. In addition, we demonstrate the extensibility of Delex by implementing a new blocking method with only 150 lines of code.

The third, and final, contribution is BigGoat, a benchmark for blocking for entity matching which mirrors how blockers are created for real-world applications. There are a wide variety of entity matching benchmark datasets. However few, if any, focus on scaling, with the majority of benchmark datasets containing fewer than 1M records. Due in part to this gap, many research blocking solutions cannot scale to large datasets, and hence are not practical for use in real-world applications. To address this problem we created the BigGoat benchmark. BigGoat consists of five realistic datasets with tables having up to 60M records. In creating BigGoat, we also develop a novel downsampling algorithm specifically designed for estimating the recall of blockers.

Collectively, the contributions presented in this dissertation represent a significant advancement in the state of the art in blocking for entity matching. In addition, this work lays the foundation for future research aimed at further improving blocking algorithms and developing more robust evaluation methodologies.

Chapter 1

Introduction

In this dissertation we study entity matching, which finds data instances that refer to the same real-world entity. We focus in particular on the blocking step of entity matching. We begin by defining the entity matching problem (EM) in Section 1.1. We discuss prior work on blocking and criteria for effective blocking in Sections 1.2 and 1.3. Next, we discuss limitations of prior work with regard to the criteria in Section 1.4. We describe the contributions of this dissertation in Section 1.5. We conclude with a roadmap of the rest of the dissertation in Section 1.6.

1.1 Entity Matching

The *entity matching* problem is to find records which refer to the same real world entity. Formally, given two table A and B with the same schema, find all pairs $(a, b) \in A \times B$ that "match", where two records $a \in A, b \in B$ "match" if they refer to the same real world entity.

Example 1.1.1. *Figure 1.1 shows a tiny example of entity matching. Table A and table B contain records referring to people. Examining these records we can see that ('Dave Smith', 'Madison', 'WI') and ('Dave D. Smith', 'Madison', 'WI'), as well as ('Dan Smith', 'Middleton', 'WI') and ('Daniel W. Smith', 'Middleton', 'WI') refer to the same person. The goal of entity matching is to find all such pairs between tables A and B.*

Typically, entity matching is done in two steps, blocking and matching.

Blocking Step: The goal of the blocking step is to reduce the number of pairs processed in the matching step while retaining as many true matches as possible. The blocking step takes as input two tables A and B and outputs a set of candidate pairs $C \subseteq A \times B$, where $|C| << |A \times B|$. By

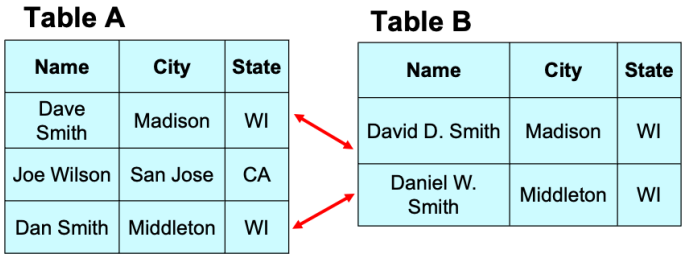

Figure 1.1: An entity matching example.

performing the blocking step we avoid running our matching algorithm on the cross product of A and B, and greatly increase the scalability of the end-to-end system.

Matching Step: The goal of the matching step is to accurately predict which records pairs match and which don't. The matching step takes as input a set of candidate pairs $C \subseteq A \times B$, which is the output of the blocking step, and outputs a set of predicted matches $M \subseteq C$. Typically, this is done by applying a matching algorithm $f : A \times B \rightarrow \{\textit{match}, \textit{non-match}\}$ to each pair $(a, b) \in C$ and only retaining the pairs for which $f(a, b) = \textit{match}$, that is, $M = \{(a, b) | (a, b) \in C \ \wedge \ f(a, b) = \textit{match}\}$.

Example 1.1.2. *Figure 1.2 shows a simple example of blocking and matching. In this example, our blocker outputs all pairs which have the same value in the 'State' column, creating a candidate set with four pairs* $C = \{(a_1, b_1), (a_1, b_2), (a_3, b_1), (a_3, b_2)\}$. *The matching algorithm is then applied to the candidate set to get two predicted matches* $M = \{(a_1, b_1), (a_3, b_2)\}$.

There are various challenges to the blocking and matching steps which have been addressed by previous work (e.g., [12, 25, 26, 48, 14, 53, 3, 11, 50, 55]). In this dissertation we focus solely on the *blocking* step.

1.2 Existing Blocking Methods

Over the past 30 years many different blocking methods have been created (see [12, 25, 26, 48, 14, 53, 3] for recent books and surveys). These methods include the following:

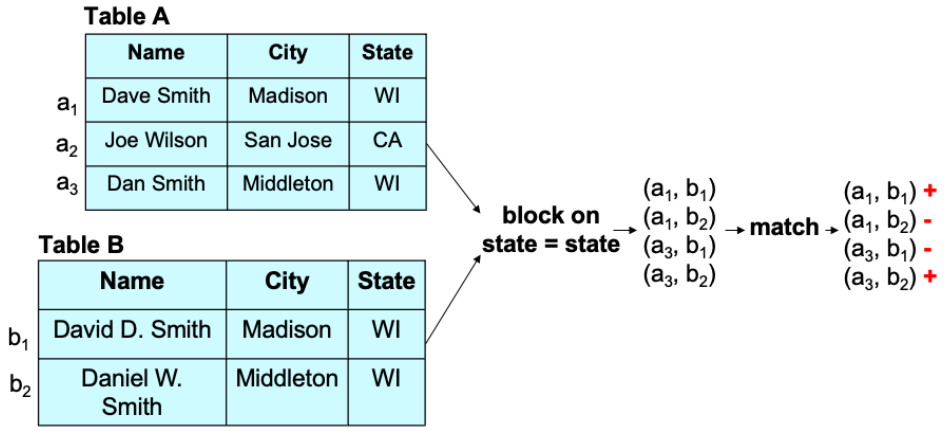

Figure 1.2: An entity matching workflow.

- **Hash:** Hash-based methods compute for each tuple one or multiple hash values (a.k.a. keys), group all tuples sharing the same hash value into a *block*, then output a pair of tuples if they belong to the same block. Examples of such methods include attribute equivalence, phonetic blocking, suffix array, etc. [11, 50, 55]. Most existing blocking methods are hash-based.

- **Sort-Based:** Sort-based blocking methods, such as sorted-neighborhood, attempt to take into account variation and misspellings in attributes by pairing records from table B with records from table A that are adjacent in a sorted order. There have been many methods developed for creating sort keys, however the simplest one is to simply sort by a single attribute [33]. For example, we could sort $A \cup B$ by the state column and then for each record $b \in B$ output b paired with the 5 records before it in sorted order and the 5 records after it in sorted order.

- **Similarity-Based:** Similarity-based blocking methods leverage similarity metrics that can be efficiently indexed. The two most common types of similarity based methods are threshold-based and top-k based [58]. Threshold-based methods take the form $f(a, b)\ op\ val$ where f is a similarity function, $f : A \times B \rightarrow \mathbb{R}$, op is a comparison, $op \in \{\geq, >, =, <, \leq\}$,

and val is a threshold, $val \in \mathbb{R}$. For example, $jaccard\_name(a, b) > .7$. Top-k based methods return the top-k values ranked using a scoring function. For example, we could rank records based on the TF/IDF score between the names and return the top-10 records from A for each record in B.

- **Rule-Based:** 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) [28]. Given a set of rules, the blocker figures out the best way to create a workflow and execute it using indexes [28].

- **Composite:** Composite blocking methods generalize rule-based blocking and can combine multiple blocking methods in a complex *pre-specified* pipeline. Examples include canopy blocking [25] and the union of a deep learning method with a rule-based method in [62].

- **Schema-Agnostic:** Schema-agnostic methods are designed to deal with data that either lacks attributes or has many sparsely populated attributes [52]. Such data is common in products where there are many possible attributes but only a few are applicable to any given product and the majority of the information in each record is contained in the product title.

- **Deep-Learning Based:** Recent work using deep learning based approaches have shown promise [62], specifically for blocking based on *semantic* similarity, as opposed to most other blocking methods, which output candidates based on *syntactic similarity*. Most of these methods consist of a deep learning mode which takes a record and outputs a d-dimensional dense vector. The cosine-similarity between the vectors is then used as a scoring function for top-k based blocking.

1.3 Effective Blocking

The effectiveness of a blocking method is evaluated using three criteria: recall, output size, and runtime.

Recall: Recall is the fraction of true matches output in the candidate set. Specifically, let $G \subseteq A \times B$ be all matches between tables A and B, and $C \subseteq A \times B$ be a set of candidates pairs output by a blocker. Recall is then defined as $|G \cap C|/|G|$. All other things being equal, higher recall is better.

Output Size: Output size is defined as the number of pairs in the candidate set output by the blocker. All other things being equal, smaller output size is better. Additionally, effective blockers have bounded output size or output size that can be predicted a priori.

Runtime: Runtime is defined as the time it takes for a blocker to produce the candidate set. All other things being equal, lower runtime is better. Additionally, effective blockers scale with additional hardware. That is, runtime can be reduced by adding more cores, RAM, and/or machines.

1.4 Limitations of Existing Work

Recall: While many blocking methods can achieve $\sim 85\%$ recall on a variety of datasets, achieving very high recall (e.g. 95%+) with existing methods in many cases is not possible. When it is possible to achieve very high recall with existing methods this typically comes at the cost of large output size or long runtimes.

Output Size: Most blocking methods can be tuned to produce a reasonable output size, but this frequently comes at the cost of low recall and/or long runtime. Additionally, most blockers cannot have their output size predicted a priori. This presents a major obstacle for using these blockers in real-world applications because running the blocker over the entire dataset can take multiple hours (or days), significantly increasing the cost of development and tuning.

Runtime: While most blocking methods can be tuned to reduce runtime, the majority of systems don't scale. The majority of blocking methods are single machine only with many being single threaded. Of the systems that have multi-machine implementations, many lack fault tolerance or have poor scale-out behavior. Specifically, many systems are not designed to tolerate worker failures or see very little speed up when adding machines to the cluster.

Because of these limitations of existing methods, there currently is no industrial-strength blocking method in academia. The goal of this dissertation is to create effective, industrial-strength methods for blocking for entity matching.

Challenges: Creating effective, industrial-strength blocking methods raises the following challenges.

- **Recall:** The first challenge when developing a blocking method is to achieve high recall. This is particularly challenging to do on a wide variety of datasets since what it means to be a match can vary significantly between contexts. For example, we might consider "iPhone 15 (red)" and "iPhone 15 (silver)" to be the same product and in other cases not. An effective blocking method should handle such variations, ideally with little or no extra configuration.

- **Scalability:** The next challenge we face in developing a blocking method is scalability with respect to dataset size and cluster size. Specifically, with respect to dataset size our method needs to have sub-quadratic runtime complexity. With respect to cluster size, we want a method with as close to linear scale out as possible. That is, if we double the number of machines in the cluster our runtime should be reduced by half.

- **Lack of Unified Execution:** The third challenge we face when developing effective blocking methods is a lack of unified execution engine. Users frequently want to combine multiple blocking methods together to take advantage of the strengths of each method. The lack of a unified execution engine means that users need to write custom code to combine methods, which is time-consuming and can lead to suboptimal efficiency.

- **Lack of Test Data:** The final challenge we face when developing an effective blocking method is the lack of test data. While there are many benchmark datasets for entity matching, the vast majority of these datasets have 1M or fewer records. These datasets are far too small for evaluating the scalability of blocking algorithms, as the data in real-world applications is frequently two or three orders of magnitude larger.

1.5 Contributions of the Dissertation

To address the above challenges, in this dissertation we make the following contributions:

Sparkly: We develop Sparkly, a novel distributed TF/IDF top-k blocking method that runs on commodity hardware with a distributed shared-nothing architecture. With Sparkly we address the challenges of recall and scalability, specifically:

- We show that Sparkly outperforms 8 state-of-the-art blocking methods across 15 entity matching datasets. In particular, Sparkly achieves higher recall with output size less than or equal to state-of-the-art methods, while having bounded output size.
- We demonstrate the scalability of Sparkly's architecture with respect to both dataset size and cluster size. For dataset size we show that Sparkly scales super-linearly but sub-quadratically on both structured and textual datasets. For cluster size we show that Sparkly has near-linear scale-out behavior when adding nodes to the cluster, that is, doubling the number of nodes in the cluster almost halves the runtime for the same data.
- We open source Sparkly with fully documented code and examples for easy reproducibility and for future work that wishes to extend Sparkly.

Delex: We develop Delex, a unified execution engine for combining multiple blocking methods in a scalable fashion. With Delex we address the challenges of scalability and lack of unified execution, specifically:

- We develop Delex, the first system built from the ground up for combining multiple blocking methods using a declarative language.
- We demonstrate the effectiveness of the Delex optimizer for reducing the runtime of blocking programs across a variety of blocking programs and datasets.
- We show the scalability of Delex with respect to cluster size and dataset size. In particular, we show that Delex has near-linear scale-out behavior and that Delex scales with the complexity of the predicates in the blocking program.

- We demonstrate the effectiveness of Delex's chunking algorithm for running very large datasets by executing a blocking program over a table of 156M records on a 10-node cluster.

BigGoat: We develop BigGoat, a novel benchmark for blocker scalability that mirrors the way blockers are created in real-world applications. With BigGoat we address the challenge of lack of test data, specifically:

- We create the first benchmark that is specifically focused on testing the scalability of blockers for entity matching.
- We present a novel down sampling algorithm designed for blocker development. In particular, our down sampling algorithm produces samples that can be used to estimate the recall of a blocker on the full dataset.
- We present a novel upsampling algorithm and give insights into the challenges of upsampling for blocker scalability testing. Specifically, we describe properties of datasets that affect the runtime of blockers and how these properties present significant challenges for creating a general purpose upsampling method that can be used to estimate the runtime of blockers.

1.6 Roadmap

The rest of this dissertation is structured as follows. Chapter 2 presents Sparkly, a novel scalable solution for blocking for entity matching. Chapter 3 presents Delex, a scalable system to combine multiple blocking methods. Chapter 4 describes BigGoat, our scaling benchmark for blockers. Chapter 5 concludes the dissertation.

Chapter 2

Sparkly: A TF/IDF Blocker for Entity Matching

In this chapter we describe Sparkly, a TF/IDF blocker for entity matching. We begin by motivating Sparkly (Section 2.1). Next, we describe Sparkly and provide rationales for key design decisions (Section 2.2). We then compare Sparkly to current state-of-the-art blockers, evaluate the scalability of Sparkly, and evaluate the stability of Sparkly's components (Section 2.3). Finally, we discuss Sparkly's performance and future research directions (Section 2.4).

2.1 Introduction

In the past few years, as a part of the Magellan project at UW-Madison, which develops an open-source EM platform [37], we have implemented many blocker types, develop new blocker types [62], and applied them to real-world EM tasks in domain sciences and industry [30]. 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" [12] and several recent EM surveys [26, 11, 50, 55] briefly discuss only a single TF/IDF solution proposed 20 years ago [46]. This solution runs on a single machine, is difficult to scale (see Section 2.5), and is shown experimentally to perform worse than other solutions [11]. 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 chapter 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 [58, 62, 47]. The 8 blockers are: 2 deep learning (DL) blockers [62], 1 blocker combining a DL blocker and a rule-based blocker [62], 3 token-based blockers from the well-known JedAI open-source EM platform [54, 56], 1 kNN blocker identified as highly promising by a recent work [58], 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 2.3 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 [24, 23, 8], which allows Lucene to perform top-k search very fast [32]. 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 chapter 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.

The page [57] provides the Sparkly code and all experiment datasets (except Hospital, which is private).

2.2 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.

2.2.1 The TF/IDF Family of Scoring Functions

TF/IDF is a well-known family of scoring functions for ranking documents in KWS [45]. 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 [25] 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:

$$s(D, Q) = [\sum_t V_D(t) \cdot V_Q(t)]/[\sqrt{\sum_t V_D(t)^2} \cdot \sqrt{\sum_t V_Q(t)^2}], \tag{2.1}$$

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 [60]:

$$s(D, Q) = \sum_{t \in Q} \frac{tf(t, D) \cdot (k_1 + 1)}{tf(t, D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{avgdl})} \cdot idf(t), \tag{2.2}$$

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 2.2). 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 2.2. Here parameter b controls how much the longer document should be penalized. The article [60] provides a detailed explanation of the intuition behind BM25, which has been shown to work quite well for KWS [45, 32].

2.2.2 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 [32]. The latest releases of Lucene, since 2015, have used state-of-the-art techniques in KWS to be both accurate and fast [32].

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* [24, 23, 8]. 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.

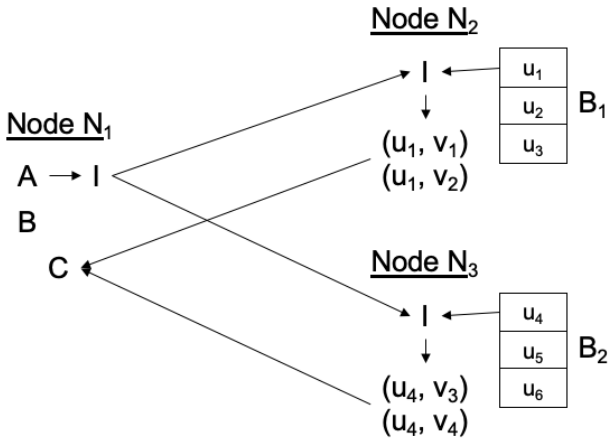

Figure 2.1: Sparkly's execution on a 3-node cluster.

2.2.3 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 (Section 2.2.5 discusses how we handle tables with different schemas and 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 2.1. (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 2.1), 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 2.1). 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 2.1 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 2.1). 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 2.1 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 2.4). 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 2.3.

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

$$t(Sparkly) = \frac{|A|}{m} t_{index} + t_{shipI} + \frac{|B|}{pnm}(t_{query} + t_{shipR}), \tag{2.3}$$

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 2.3).

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.

2.2.4 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.

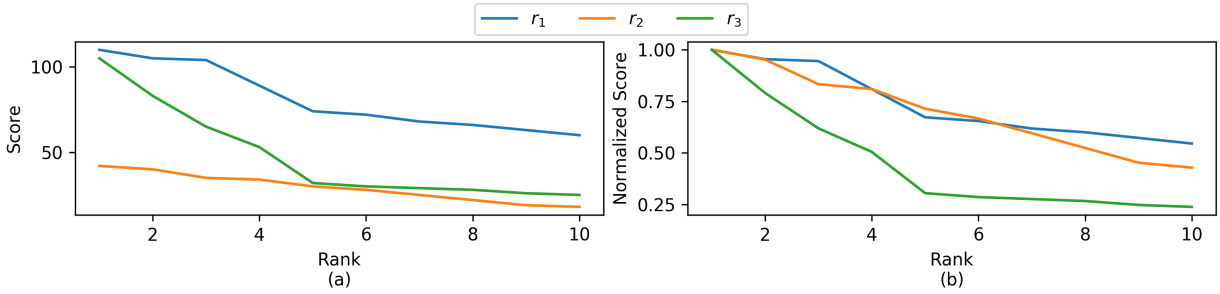

Figure 2.2: Illustrating the discriminativeness of configs.

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 2.2.1. 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 2.2.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 2.2.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 2.2.a, this is indeed true for r_3 and r_1: $AUC(r_3) < AUC(r_1)$ 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 2.2.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

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

To understand the above formula, imagine we have plotted the curve of the scores $s_1, \ldots, s_{k'}$, 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 $\frac{1}{k' \cdot s_1}$.

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 [64] 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 instead 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 $\hat{R}$.

2.2.5 Blocking for Deduplication and Tables with Different Schemas

We now discuss how Sparkly handles 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, last-name = 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.

2.3 Empirical Evaluation

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 2.1: Datasets for our experiments.

Datasets: We use 15 datasets described in Table 2.1, which come from diverse domains and sizes, and have been extensively used in recent EM work [58, 62, 47, 43] (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 [47] 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 2.3.5 and Section 2.4.

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

Autoencoder, Hybrid, Union(DL,RBB): A recent work [62] 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 [62]. The work [62] 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: JedAI [54, 56] 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).

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.

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.

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.

kNN-cosine, kNN-jaccard: Finally, a recent work [58] 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.

2.3.1 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).

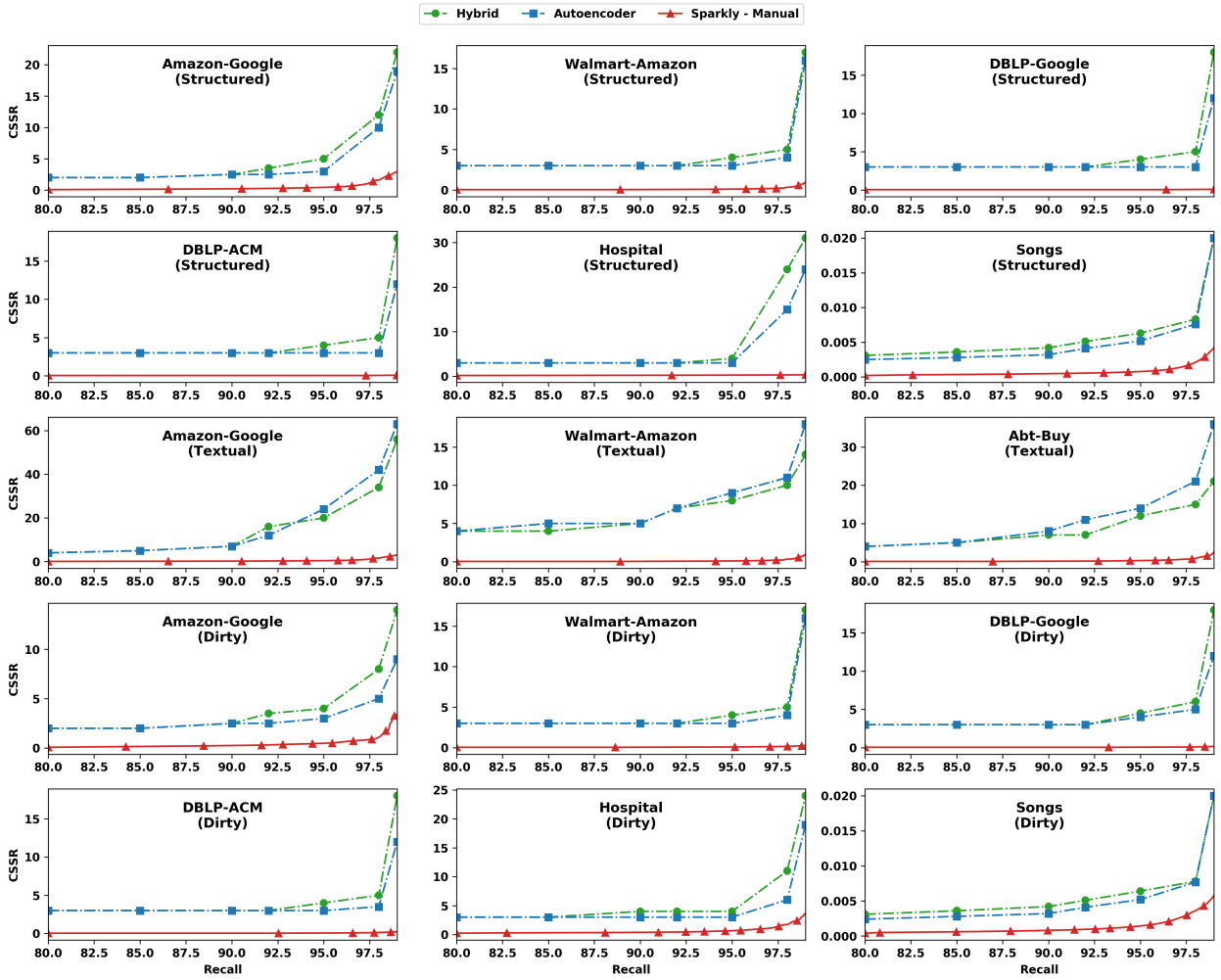

Figure 2.3: SM vs. the two best DL methods in terms of recall and blocking output size.

Comparing SM to DL Methods: Figure 2.3 compares SM with the two best DL blockers, Autoencoder and Hybrid [62]. 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.

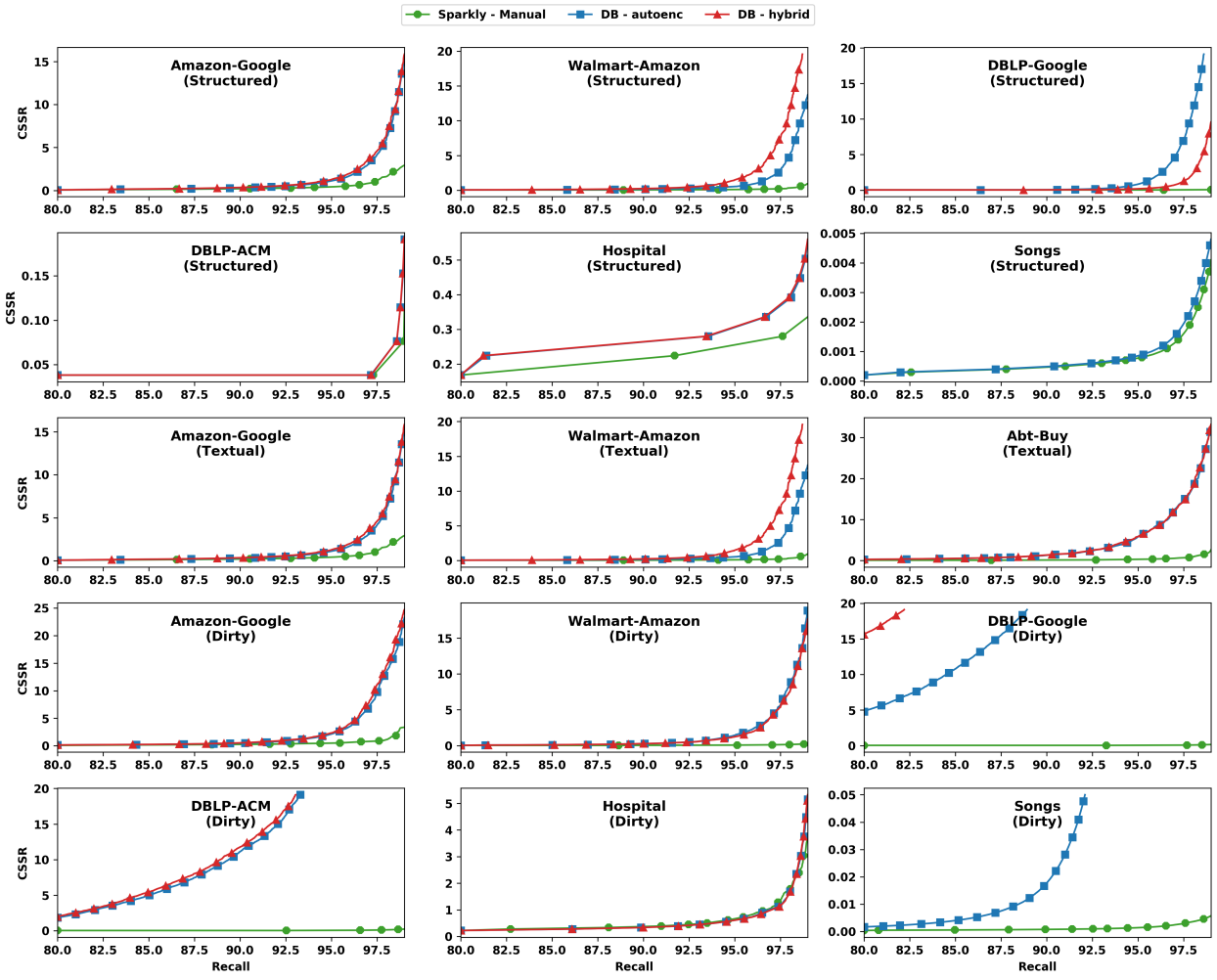

Figure 2.4: Comparing SM to DL methods which use the same attributes as SM to block; here "DB - autoenc" and "DB - hybrid" refer to the Autoencoder and Hybrid methods of the DeepBlocker paper [62], respectively.

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 2.4 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.

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

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 2.2.

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%.

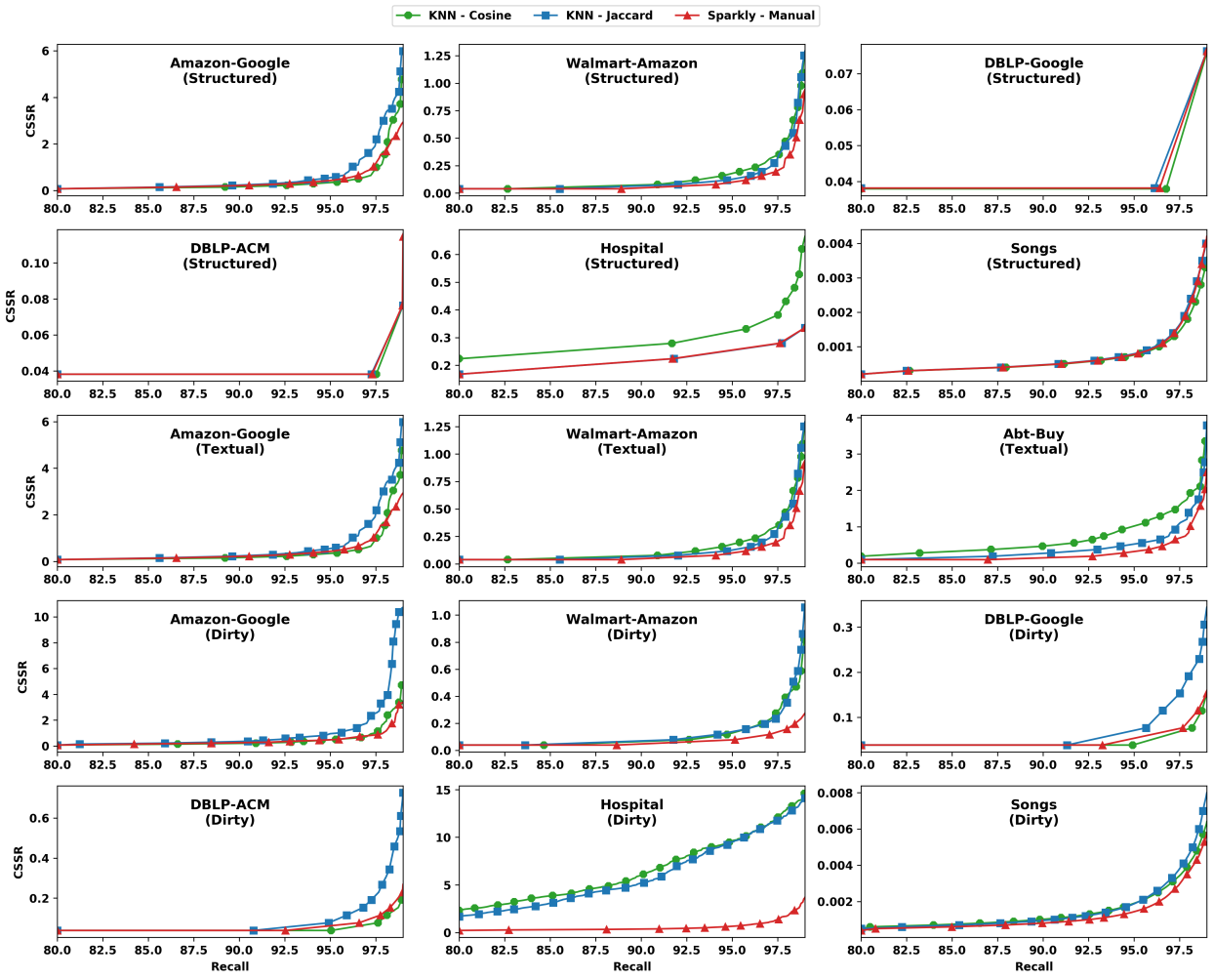

Figure 2.5: SM vs. kNN-Cosine (5-gram) and kNN-Jaccard (3-gram) in terms of recall and blocking output size.

Comparing SM to kNN Methods: Finally, we compare SM with the kNN methods. As mentioned earlier, a recent paper [58] 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 2.5 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).

When comparing kNN-Jaccard to SM, Figure 2.5 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 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 2.6 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 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.

It is interesting that SA is worse than SM on some datasets. 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).

2.3.2 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 2.7.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 [59] 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.

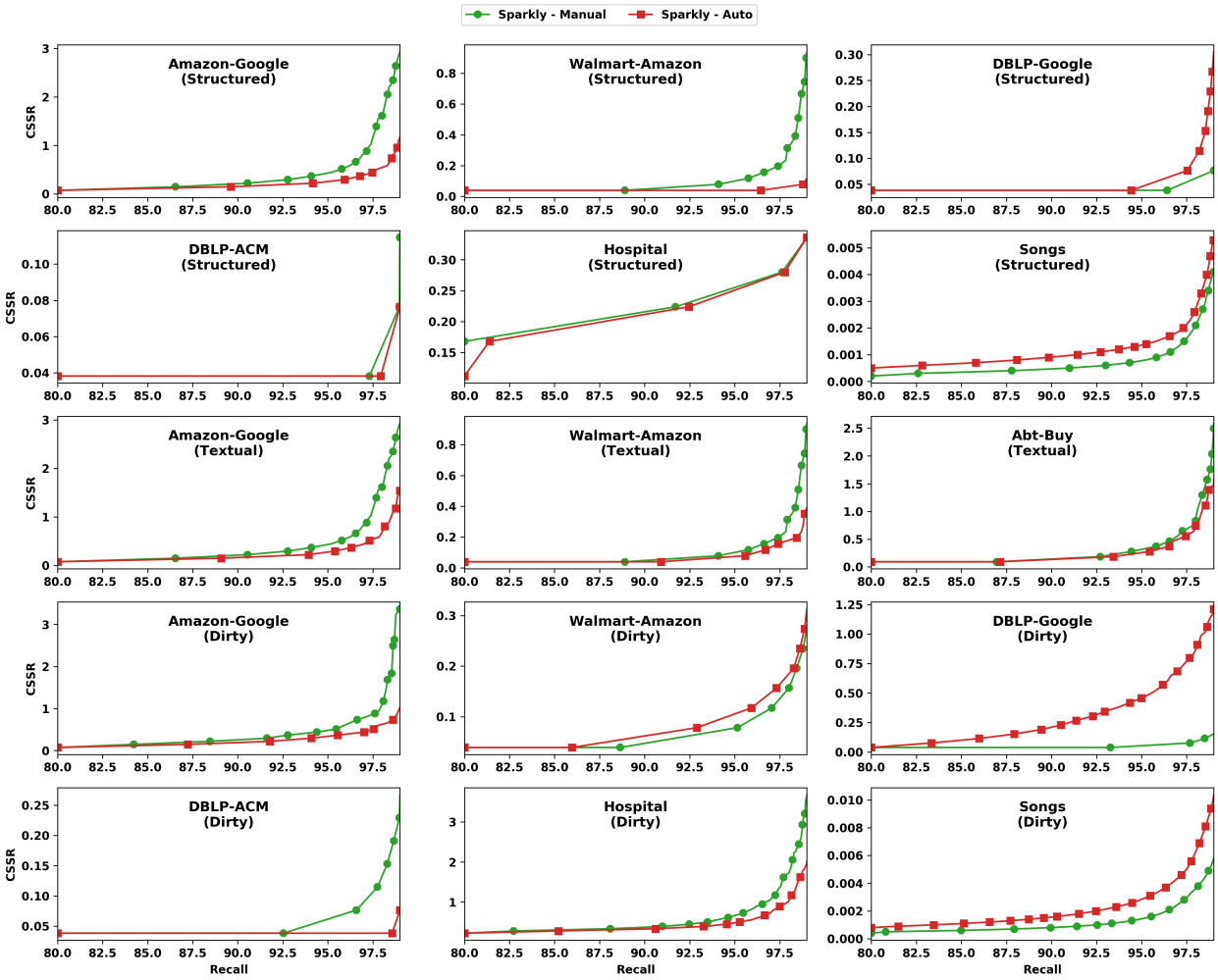

Figure 2.6: SM vs. SA in terms of recall and output size.

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.

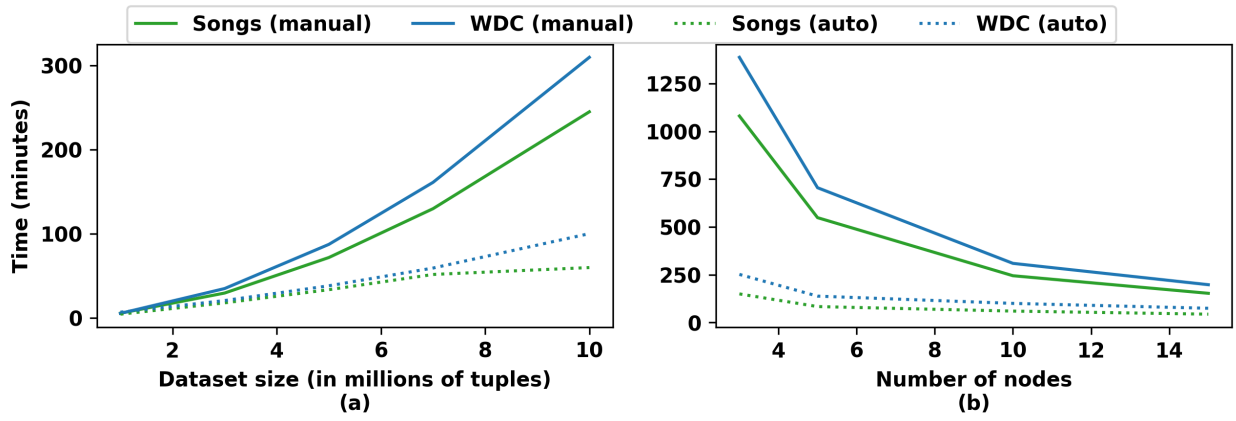

Figure 2.7: Runtime for (a) varying dataset sizes, and (b) varying cluster sizes, using datasets of 10M tuples each.

Figure 2.7.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 2.3.5.

2.3.3 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 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.

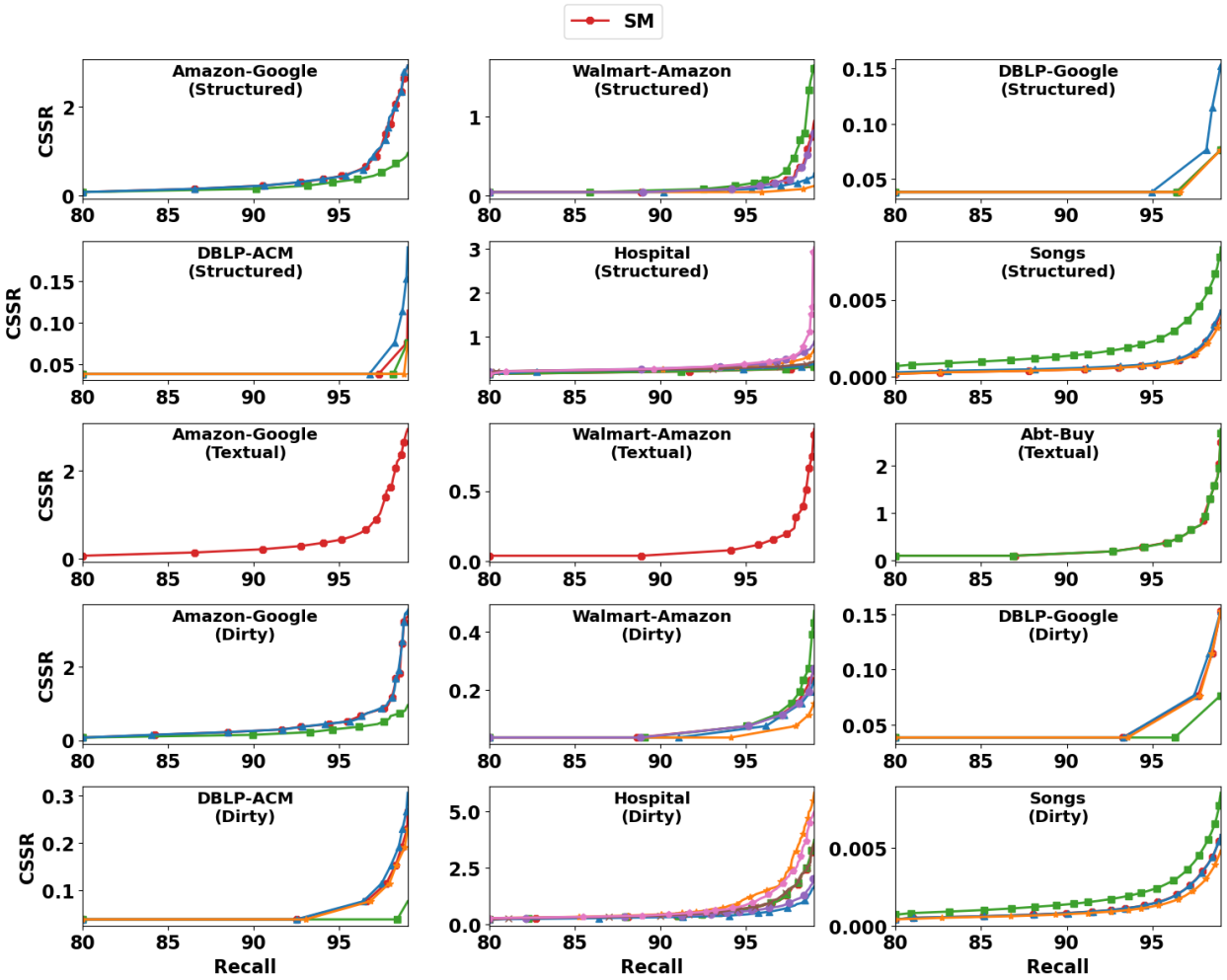

Figure 2.8: Performance of SM for varying sets of blocking attributes.

2.3.4 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 2.8 examines varying S. Consider the first plot, Amazon-Google. Here SM blocks on

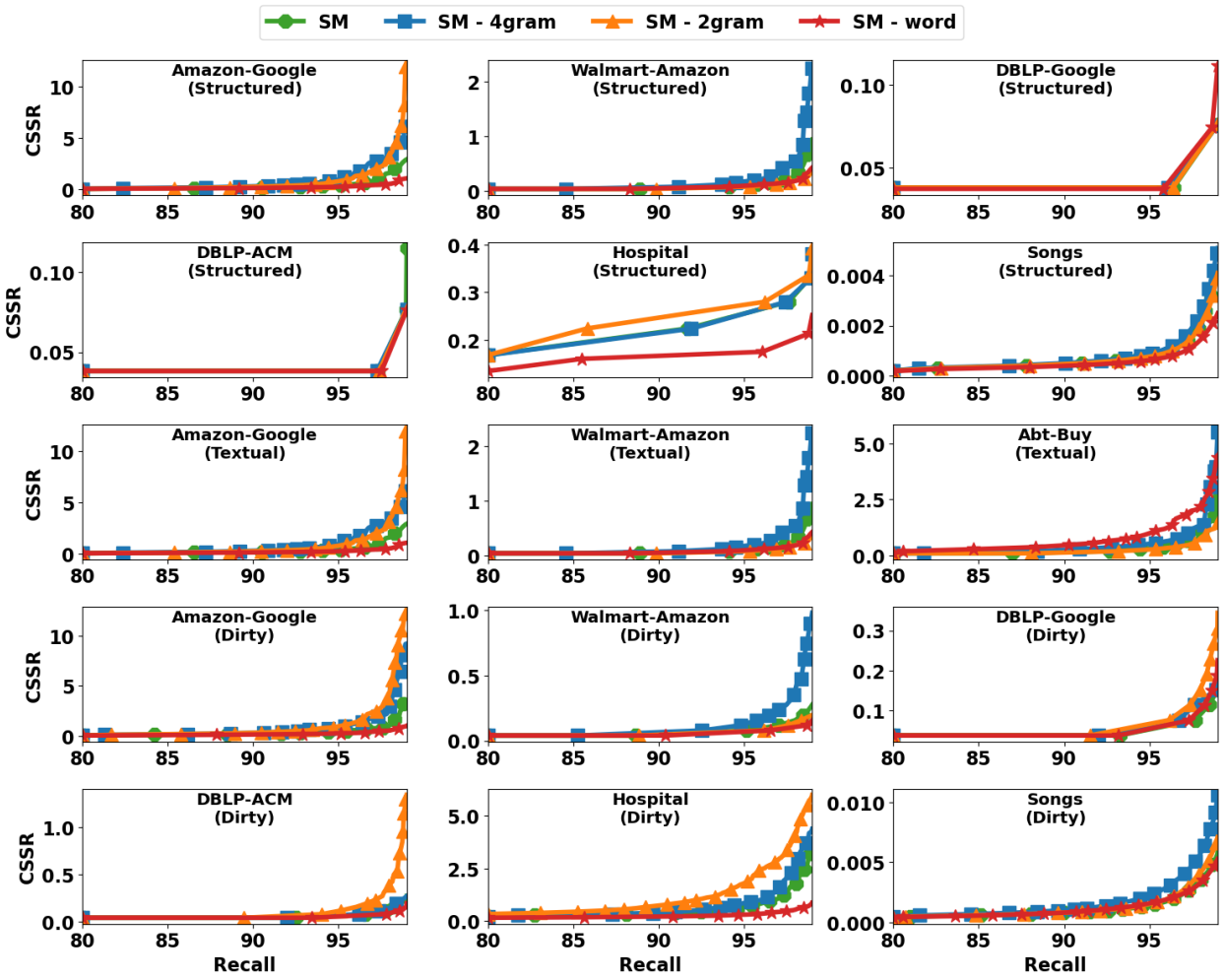

Figure 2.9: Performance of SM for different tokenizers.

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 2.8 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 2.9 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.

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. Figure 2.10 and Figure 2.11 show the results of varying k_1 and b, respectively. 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).

Varying (1) from 5K to 15K changes SA minimally (Figure 2.12). Varying (2) from 200 to 300 again changes SA minimally (only up to 0.2% CSSR on 1 dataset) (Figure 2.13). Similarly, varying (3) from 8 to 12 and varying (4) from 2 to 4 show minimal changes(Figure 2.14 and 2.15). In all cases, the default values for (1)-(4) provide a good curve, which is either the best or very near the best.

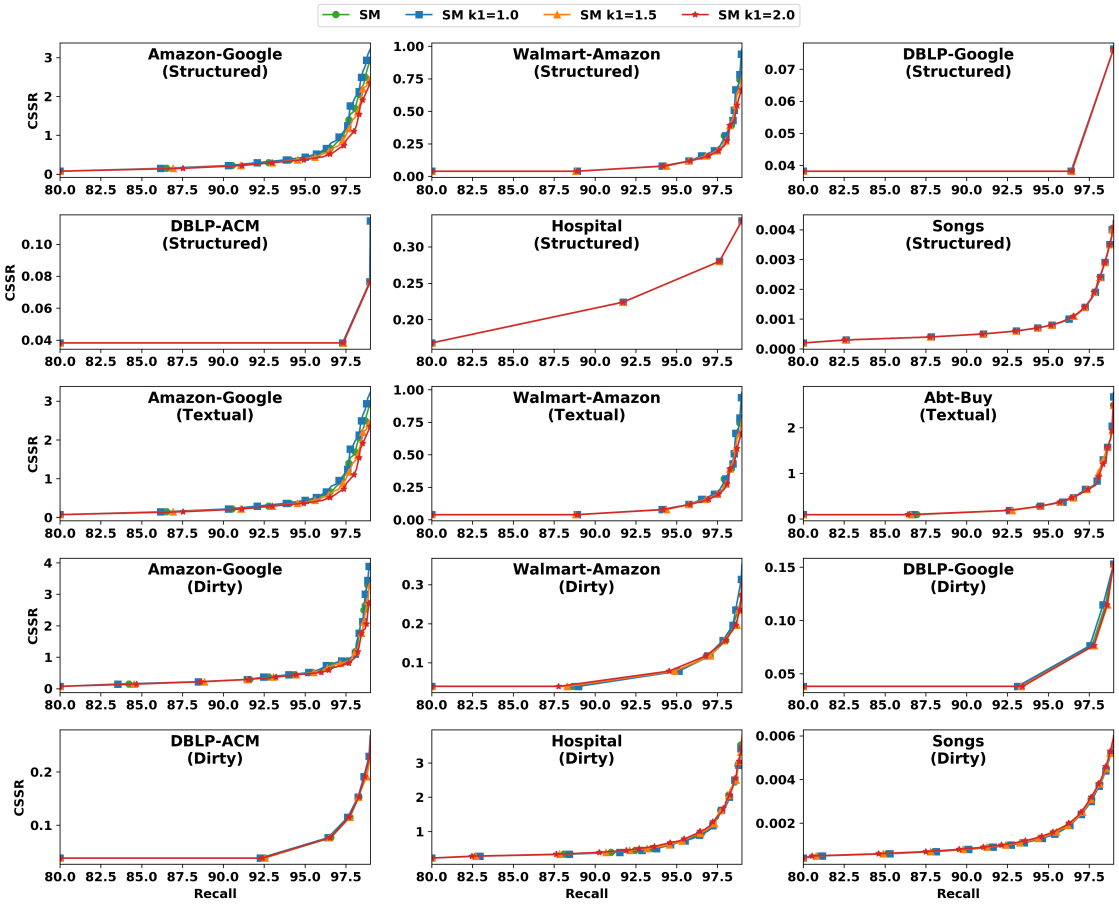

Figure 2.10: Varying the parameter k_1 of BM25.

2.3.5 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 and has complete gold [21]. MB (Music Brainz) blocks a table of 20M songs against itself and comes with the complete gold, but this gold is synthetic. A copy of MB is available at

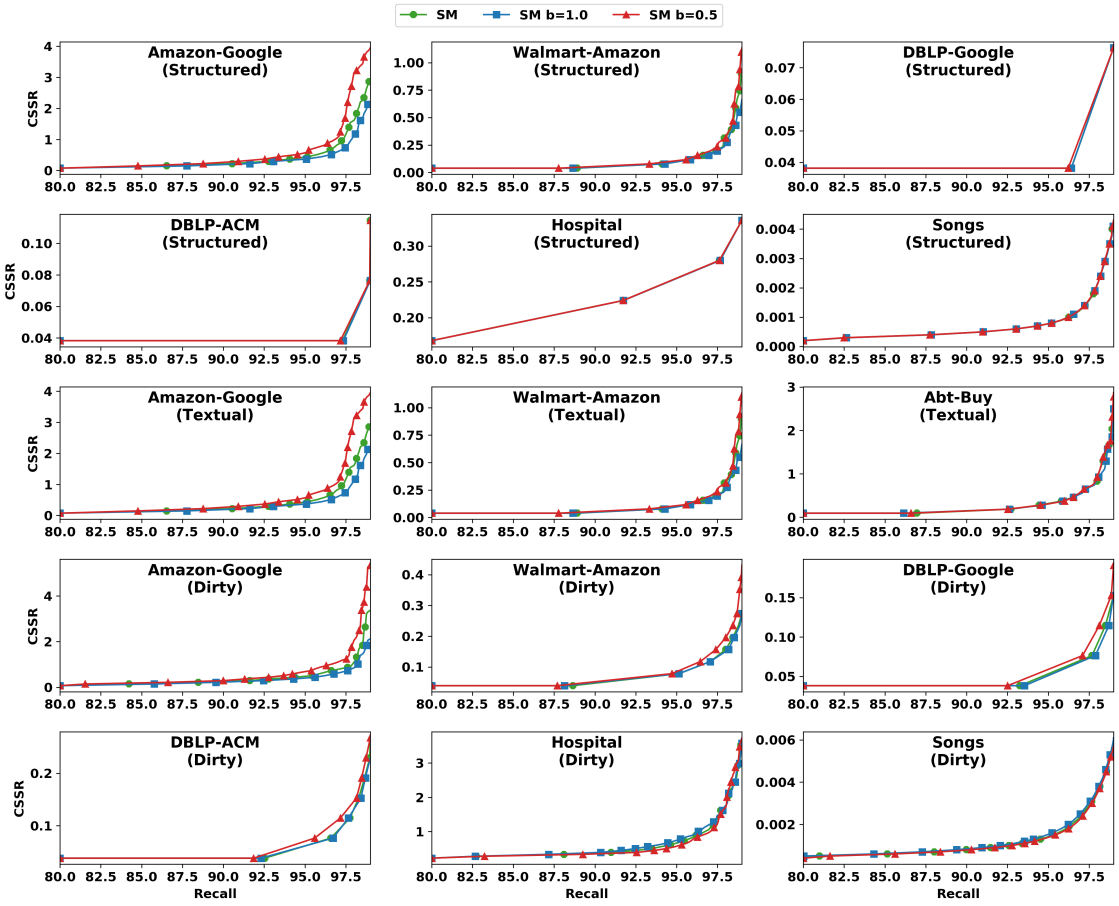

Figure 2.11: Varying the parameter b of BM25.

[1]. WDC is a single table of 26M tuples, each describing a product. WDC is available at [59]. It comes with some gold matches, but not the complete gold.

Companies is a document data set used by a prior work [47]. 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.

Table 2.16 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 "/". Column "Time"

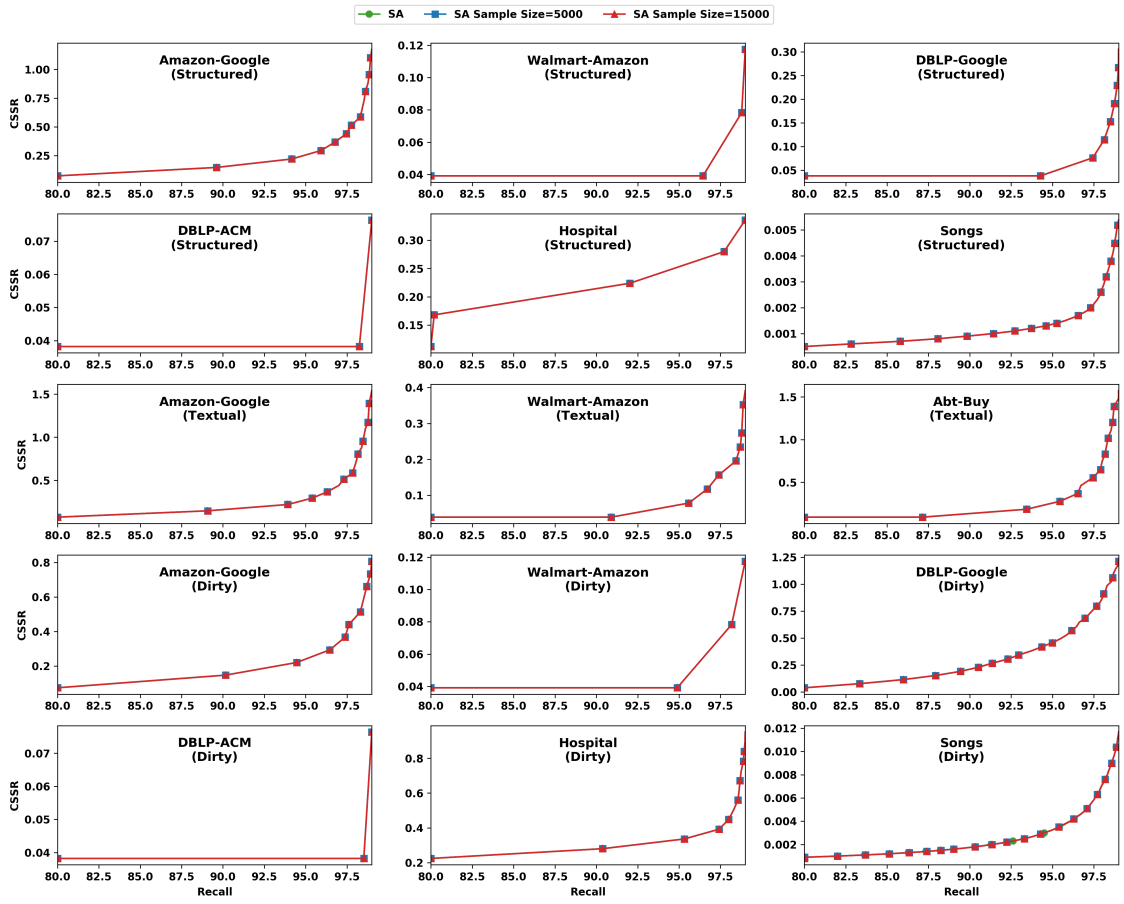

Figure 2.12: Config searcher: varying the size of B', a sample of table B on which to score the configs; default value is 10K.

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 2.16 (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 [62], for datasets of up to 1M tuples. It runs on a

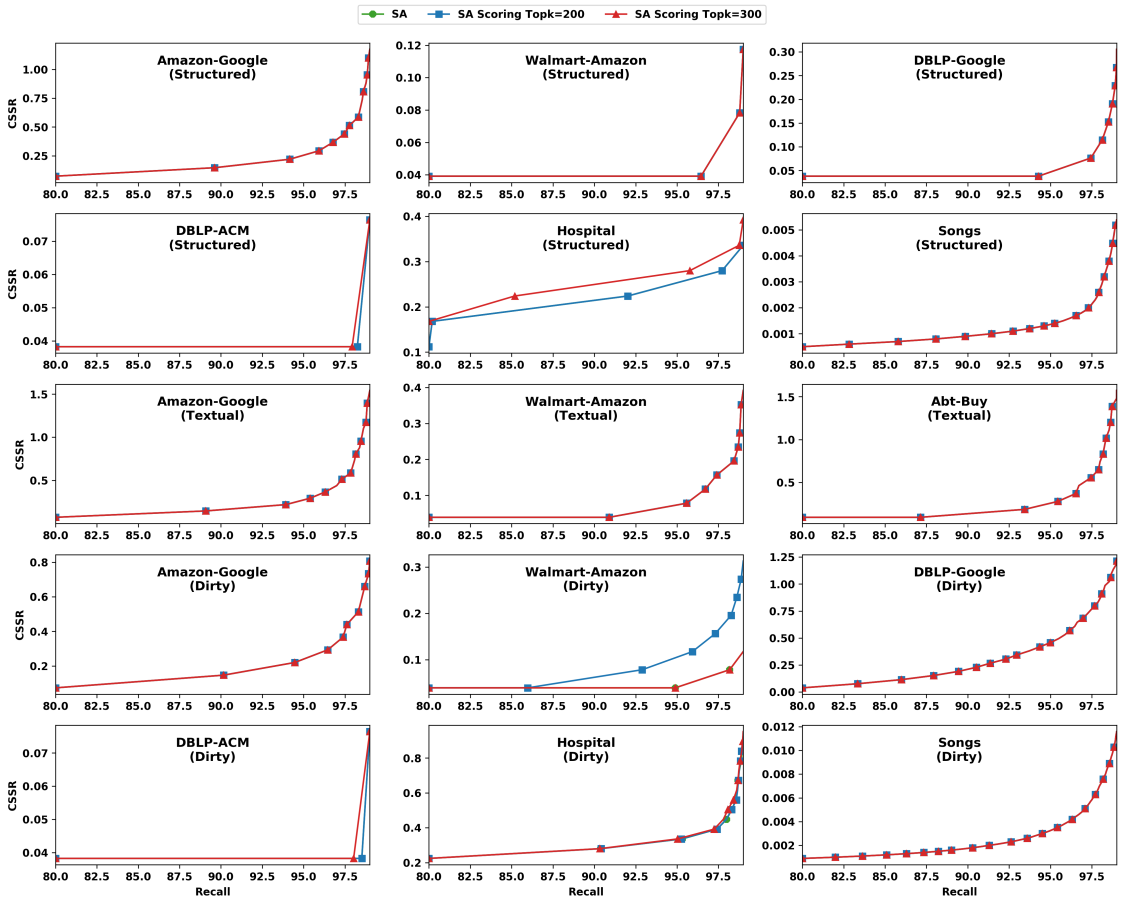

Figure 2.13: Config searcher: varying the number of tuples returned in each querying; default value is $k = 250$.

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 2.16 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

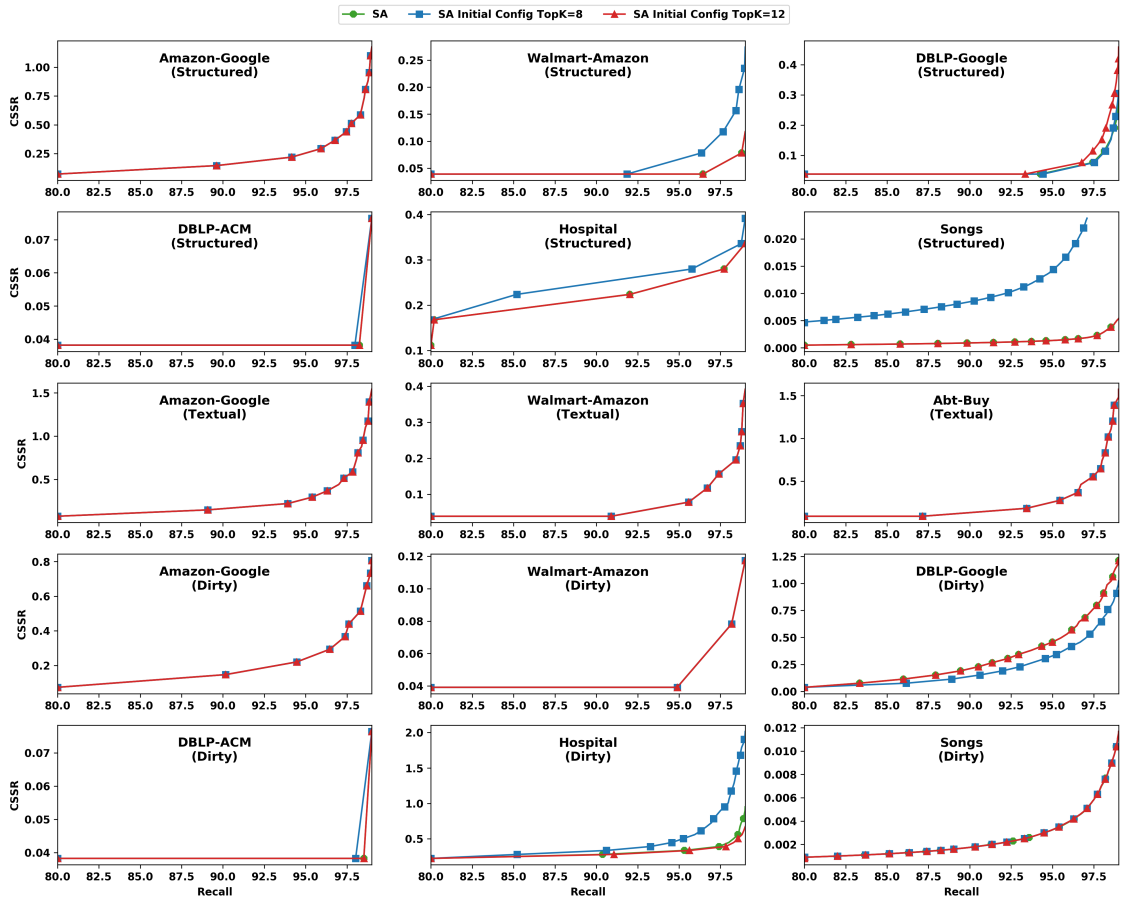

Figure 2.14: Config searcher: varying the number of initial configs selected for the search; the defaul value is 10.

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.*

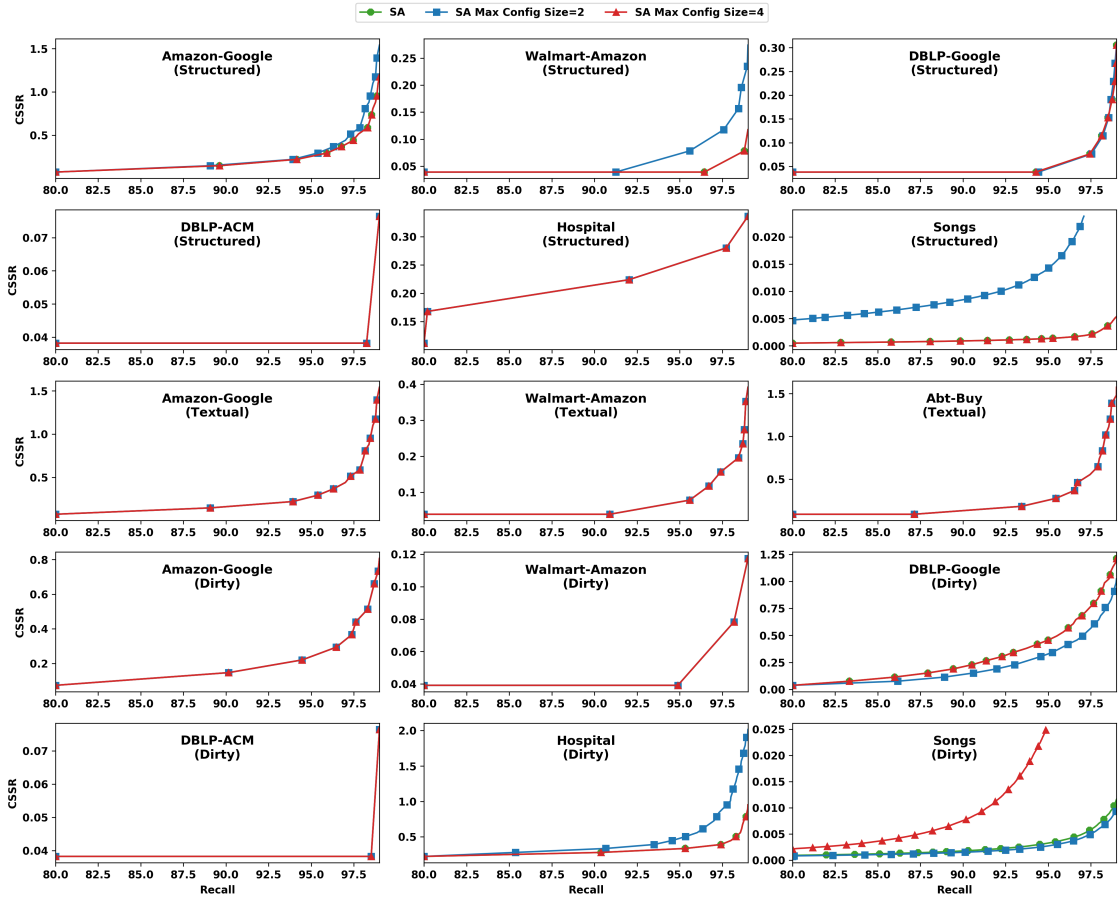

Figure 2.15: Config searcher: varying the max number of attributes considered in a config; the default value is 3.

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 2.16: Applying Sparkly and DL methods to large datasets.

2.3.6 Scalability of Blocking Methods

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

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 [66]). As discussed in Section 2.5, many works have developed solutions for top-k similarity joins, including support for Jaccard [66]. 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 [66] 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 are not aware of any system which has implemented the WAND or block-max WAND algorithm for set cosine.

2.4 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 [15]. It is not clear why they have received no attention in the blocking step (except several works

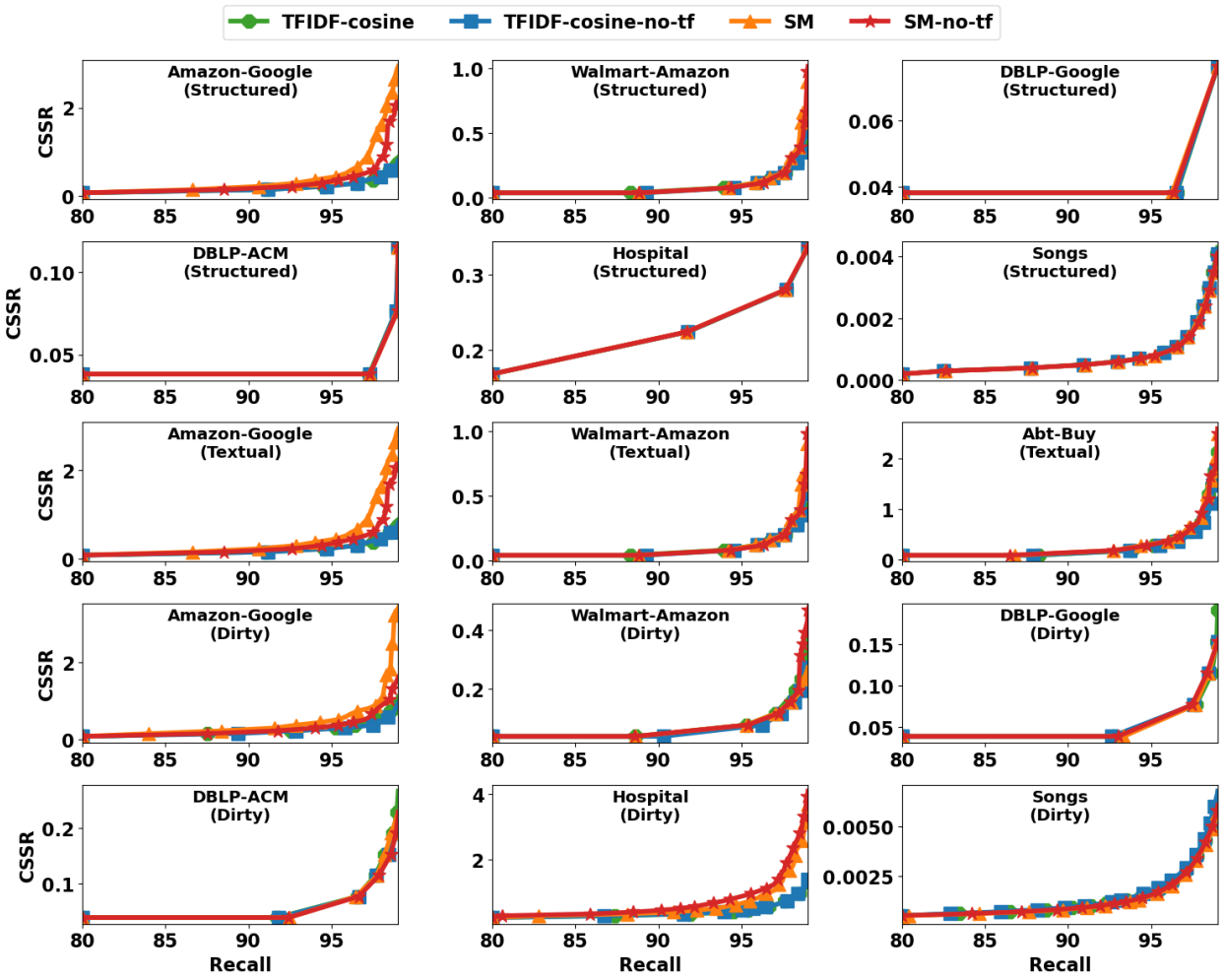

Figure 2.17: The effect of removing IDF from SM and TFIDF-cosine.

as discussed in Section 2.5). 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 [11, 50, 55]. 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 [11, 50, 55]. 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 [32]. 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.

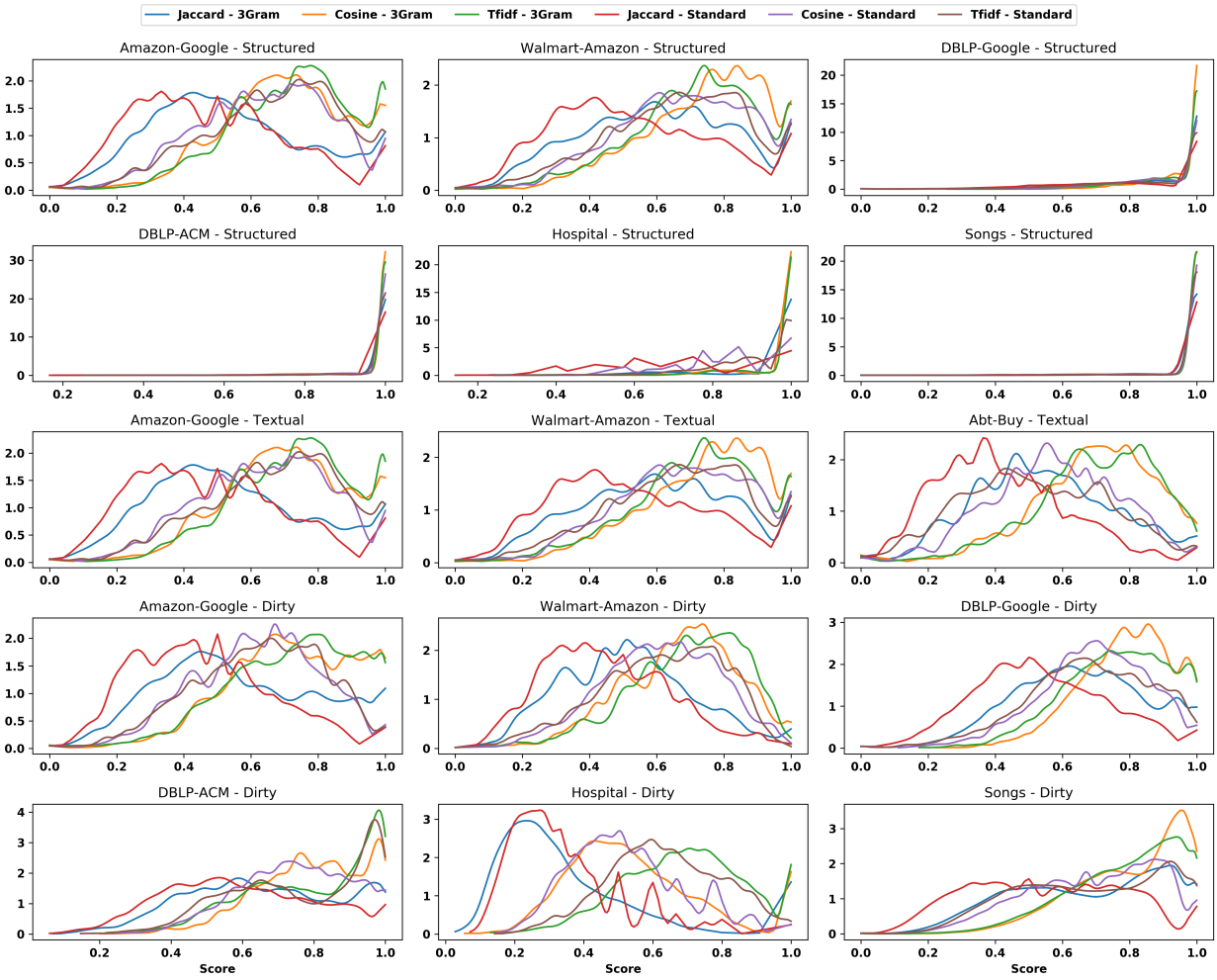

Figure 2.18: Distributions of the scores of gold matches.

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 2.18 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 2.18 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 2.3 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 2.17 shows the results when we remove IDF from SM (by dropping "idf(t)" in the BM25 formula of Equation 2.2 in Section 2.2). The figure shows that SM greatly outperforms SM-no-idf.

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 2.2, 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.

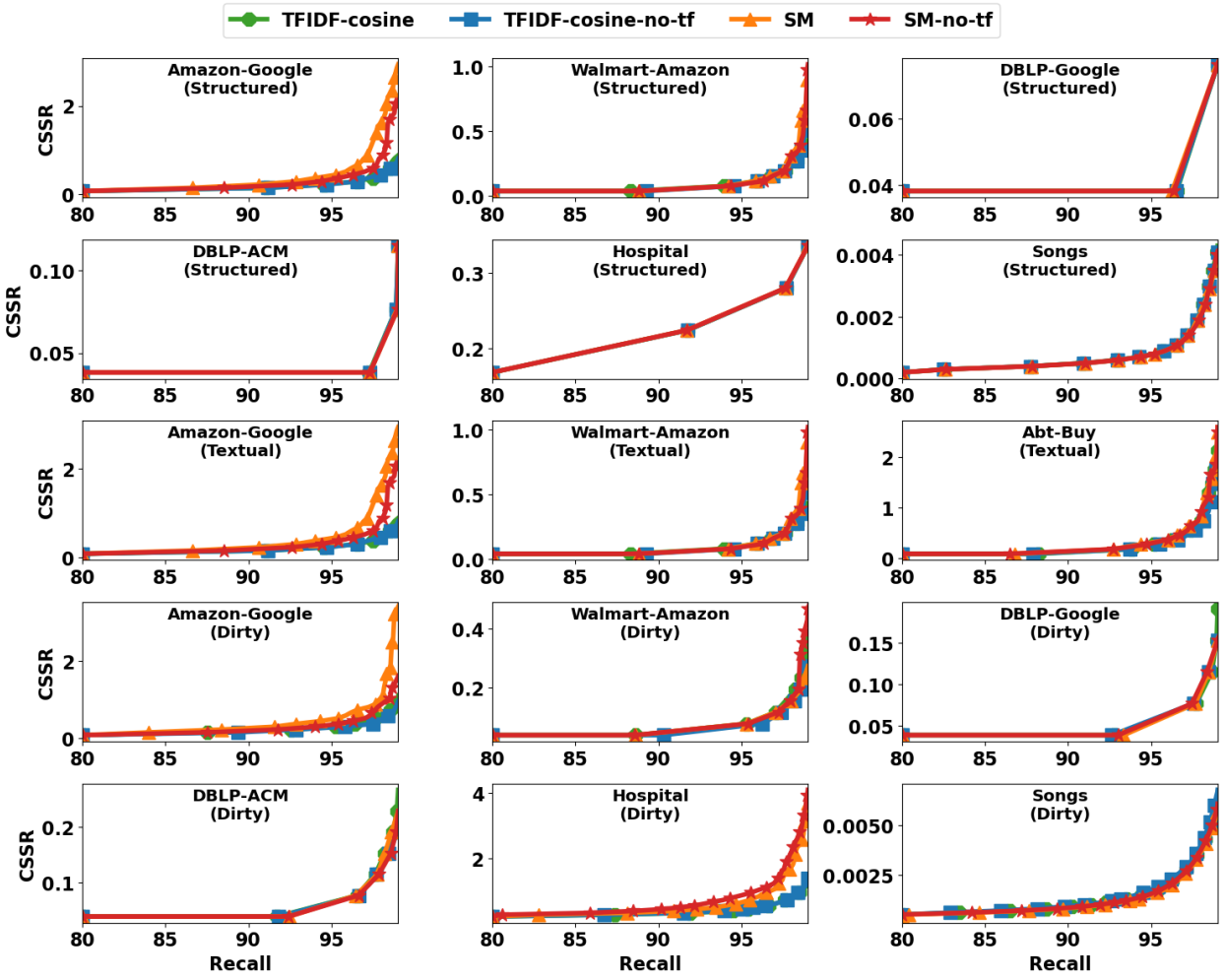

Figure 2.19: The effect of removing TF from SM and TFIDF-cosine.

Interestingly, when we performed a similar experiment where we removed TF, we did not see a clear trend. See Figure 2.19. 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 [47] to evaluate DL methods for matching). 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.

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 [9], which can be viewed as the Jaccard similarity function using IDF. SM+ is an extension of SM that we will describe shortly.

Figure 2.20 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 2.1 in Section 2.2), 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. 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 $b = 0.75$.

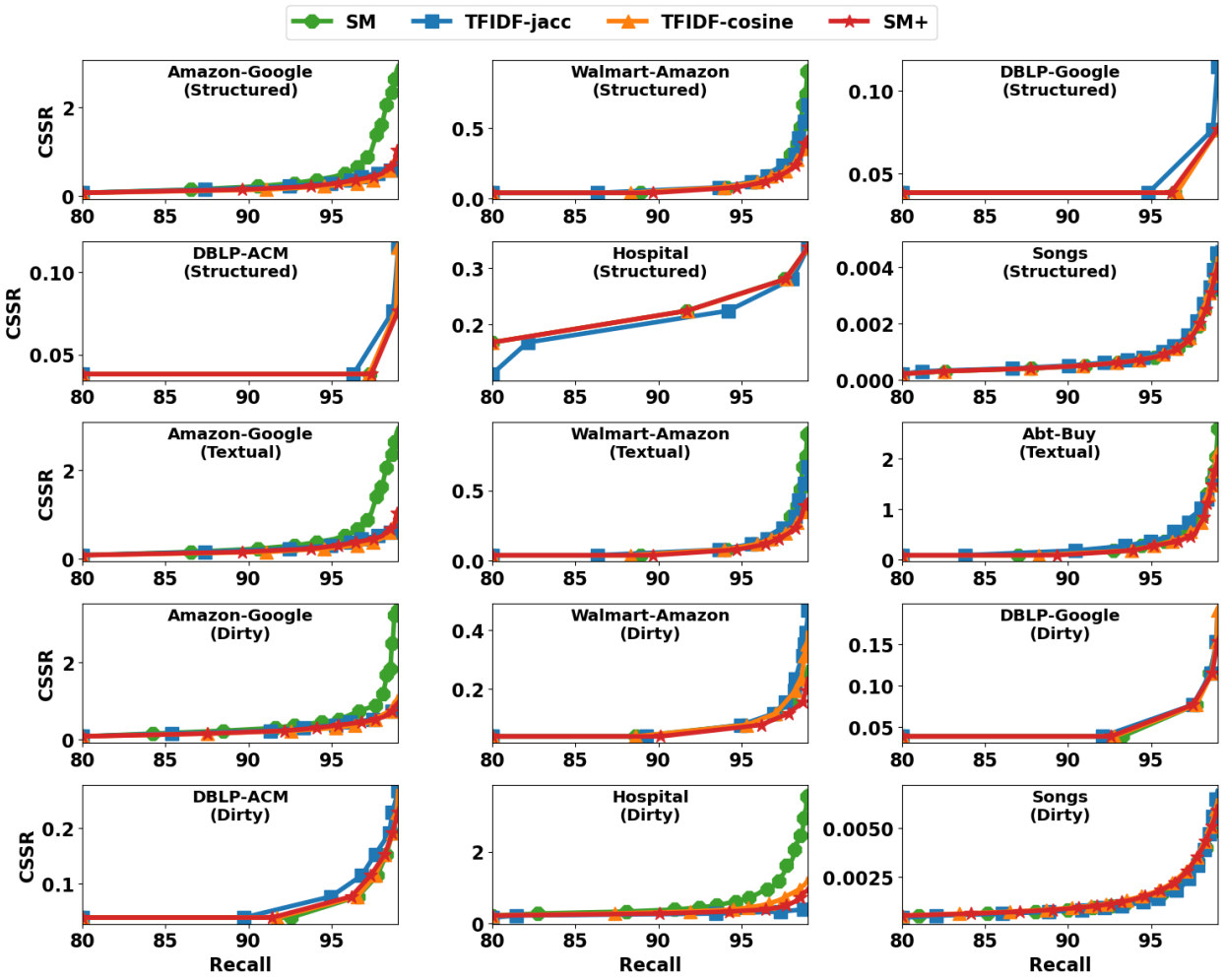

Figure 2.20: Evaluating different scoring functions.

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

We apply this new scoring function to SM to produce the SM+ solution. Figure 2.20 shows that SM+ performs very well, being either the best solution or very close to the best solution on all datasets.

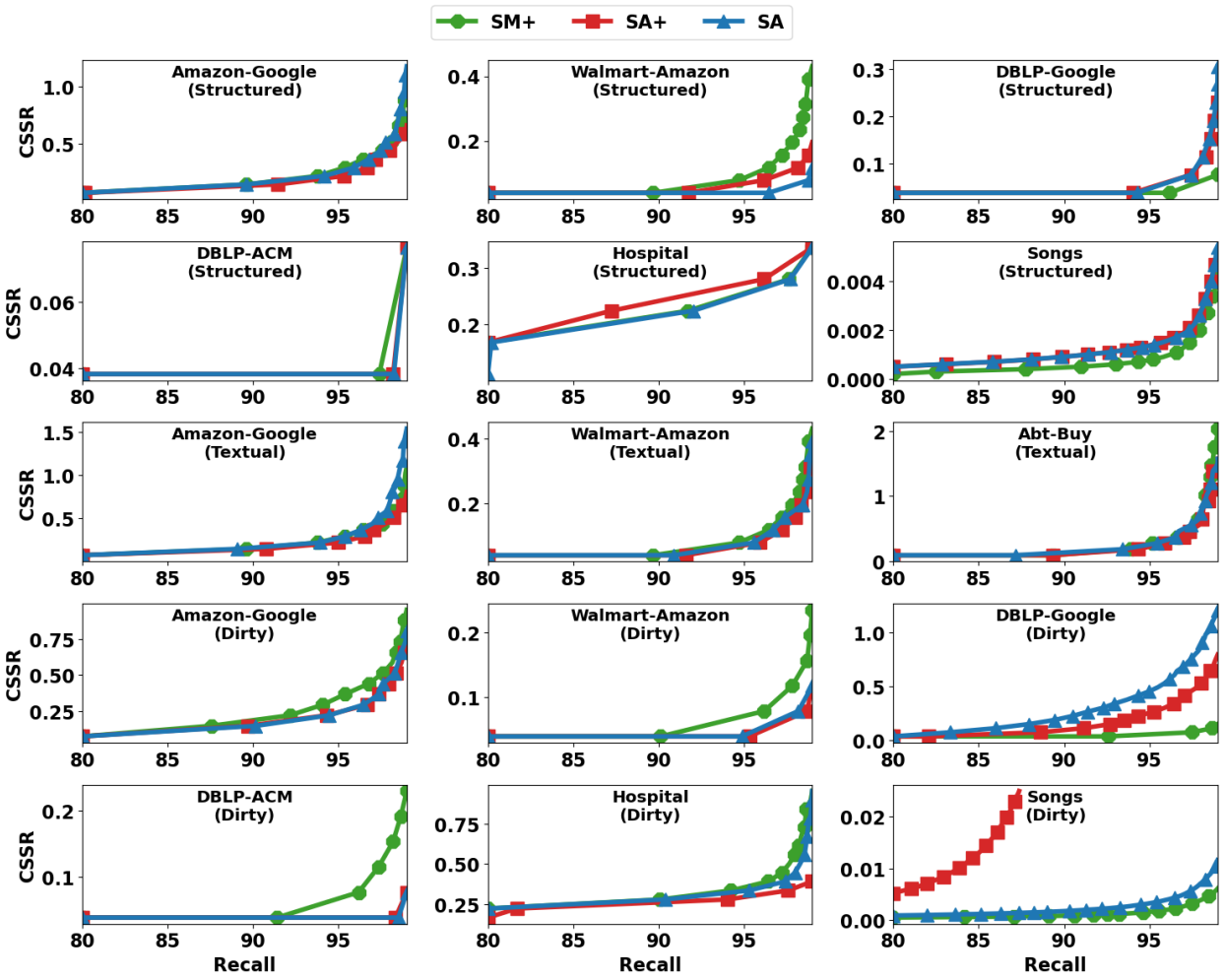

Figure 2.21: Comparing SM+, SA, and SA+.

We modified SA similarly to produce the SA+ solution. Figure 2.21 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 [47] 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. 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, 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.

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 2.3). 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.

2.5 Additional Related Work

We have discussed related work throughout this chapter. We now discuss additional related work. EM has been a long-standing challenge in data management (see [12, 25, 26, 48, 14, 53, 3] for recent books and surveys). There has been multiple academic efforts on building scalable EM systems such as JedAI [54, 56], Magellan [37], and CloudMatcher [31].

Over the past decades, numerous blocking solutions have been developed. See [26, 11, 50, 55] for surveys, and see Section 1.2 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 [46]. 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 [11] 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 [49], 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 [9] develops a scoring function that can be viewed as the Jaccard similarity function using IDF. We evaluated this function in Section 2.4.

The TF/IDF measure has long been used in IR and Web search [45]. 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 [32]. 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 [32]. TF/IDF has long been used in the matching step of EM [15].

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 [66]. 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 [67] 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 [62] 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 [42] 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., [66, 65]), 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 [61], and our Spark-based probing method for blocking is reminiscent of distributed/parallel joins for relational data [40, 39]. But here we consider the novel context of blocking for EM. Finally, the work [7] describes an industrial blocking solution at Amazon, which uses meta blocking to manage token-centric blocks and uses sophisticated techniques to scale.

2.6 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 chaper 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.

Chapter 3

Delex: Declarative Blocking for Entity Matching

As we discussed in Chapter 2, many different blockers (that is, blocker methods) have been developed in the past thirty years. Each of the these blockers comes with its own set of strengths and weaknesses, and in many use cases, no one blocker will dominate another.

This leads many real world users to want to combine blocking methods to be able to take advantage of the strengths of multiple methods at the same time. Currently, this combining is done in an ad-hoc fashion. Doing such ad-hoc combinations require trading off performance and coding effort. Tight integration typically leads to better performance but at the cost of significant coding effort, while looser integration requires less coding effort but at the cost of performance.

Our goal with Delex is to build a system from the ground up to combine multiple blocking methods, significantly reducing the coding effort while maintaining overall system performance. In particular, as we will see below, Delex will enable *declarative blocking*. Users can use a relatively simple blocking language to quickly combine multiple blocking methods. Delex will translate blocking programs written in this language into execution plans and will efficiently execute these plans on a cluster of machines.

In the rest of this chapter we will discuss Delex blocking programs, the translation, execution, and optimization of these programs, and empirical results showing the promise of the Delex approach.

3.1 Delex Blocking Programs

In Delex a blocking program $Q = (S, T)$ is a set of *keep rules* S and a set of *drop rules* T. Each rule is a conjunction of predicates, and each predicate is defined by a triple $P_i = (f_i, op_i, val_i)$. We formally define these concepts below.

Comparison Function: We define a *comparison function* f to be a function $f : A \times B \rightarrow \mathbb{R}$ which takes in two tuples $a \in A, b \in B$ and returns a real value. For example, a comparison function could compute the edit distance between the name attributes of the tuples. We currently support the following comparison functions, $jaccard$, $cosine$, $overlap\_coeff$, $edit\_distance$, $smith\_waterman$, $jaro$, $jaro\_winkler$, $exact\_match$. We chose these comparison functions because they are some of the most common string similarity measures used in research for entity matching. We also support user defined functions and $tfidf\_topk$. However we reserve the discussion of these functions for Section 3.6 and Section 3.7, respectively.

Predicate: A *predicate* is a triple $P_i = (f_i, op_i, val_i)$, where f_i is a comparison function, op_i is a comparison operator in $\{>, \geq, =, \neq, \leq, <\}$, and val_i is a real number. We say a pair $(a, b) \in A \times B$ "satisfies" a predicate $P_i = (f_i, op_i, val_i)$, written as $P_i(a, b)$, if $f_i(a, b)\ op_i\ val_i$. For example, if $P_i = (jaccard\_name, >, .7)$, then a pair (a, b) satisfies P_i if $jaccard\_name(a, b) > .7$.

Rule: A *rule* R_i is a conjunction of predicates, defined by a set of predicates $R_i = \{P_1^i, \ldots, P_{|R_i|}^i\}$. We say a pair $(a, b) \in A \times B$ "satisfies" a rule, written as $R_i(a, b)$, if the pair (a, b) satisfies all predicates in R_i. For example, if we have a rule R_i with two predicates $P_1^i = (jaccard\_name, > , .7)$ and $P_2^i = (jaro\_phone, >, .8)$, then

$$R_i(a, b) = P_1^i(a, b) \wedge P_2^i(a, b) = jaccard\_name(a, b) > .7 \wedge jaro\_phone(a, b) > .8$$

Blocking Program: A *blocking program* $Q = (S, T)$ is a set of one or more keep rules S, and a set of zero or more drop rules T. Both S and T are in *disjunctive normal form*, that is, the rules are combined using logical OR. The candidate set C that is output by executing blocking program Q over tables A and B is all pairs $(a, b) \in A \times B$ such that (a, b) satisfies at least one keep rule in

S and none of the drop rules in T. Put in terms of set notation:

$$\begin{aligned} C &= Q(A,B) \\ &= \{(a,b) \mid (a,b) \in A \times B \wedge \exists R \in S, R(a,b) \wedge \nexists R' \in T, R'(a,b)\} \end{aligned}$$

For example the following blocking program has two keep rules and a single drop rule:

$$\begin{aligned} Q = (\{ & \\ & \{(jaccard\_name, >, .7), (jaro\_phone, \geq, .8)\}, \\ & \{(jaccard\_name, >, .7), (edit\_dist\_phone, >, .8), (jaro\_zipcode, \geq, .9)\} \\ & \}, \\ & \{ \\ & \{(cosine\_zipcode, <, .4)\} \\ & \}) \end{aligned}$$

Q is equivalent to the following boolean formula:

$$\begin{aligned} Q(a,b) = ((&jaccard\_name(a,b) > .7 \wedge jaro\_phone(a,b) \geq .8) \\ &\vee (jaccard\_name(a,b) > .7 \wedge edit\_dist\_phone(a,b) > .8 \wedge jaro\_zipcode \geq .9)) \\ &\wedge \neg(cosine\_zipcode(a,b) < .4) \end{aligned}$$

3.2 Translating Blocking Programs

We now discuss translating a Delex blocking program, as described earlier, into an execution plan. First we define the notion of an execution plan consisting of operators. Then we discuss how to preprocess a Delex blocking program and translate it into a default execution plan (later we discuss how to optimize this plan).

In what follows, we use $a.x$ to refer to the field x of the tuple $a \in A$. Additionally, we use $A.x$ to refer to x column of A, that is, $A.x = \{a.x | a \in A\}$.

3.2.1 Execution Plan

We first define the notion of execution plan.

Candidate Pair: We define a "candidate pair" to be $(a.id, b)$ where $a.id$ is the id of tuple $a \in A$ and $b \in B$. Additionally, we say a candidate pair $(a.id, b)$ "satisfies" predicate P if (a, b) satisfies P, that is, $P(a, b) = true$.

Index: An index $I_P(A)$ is a data structure for which we have two functions $build$ and $lookup$. Function $build$ takes as input a predicate P and a table A, and returns an index $I_P(A)$. That is, $build(P, A) = I_P(A)$. $lookup$ takes as input an index $I_P(A)$ and tuple $b \in B$, and returns a superset of the ids of tuples in A that satisfy predicate P with b. Formally, $lookup(I_P(A), b) \supseteq \{a.id | a \in A \wedge P(a, b)\}$.

Indexable Predicate: We define an "indexable predicate" to be a predicate P for which we have both a $build(P, A)$ and $lookup(I_P(A), b)$ function provided.

Hash Table: In the Delex context, a hash table is a data structure for which we have two functions $build\_hash\_table$ and get. $build\_hash\_table$ takes as input a table A and returns a hash table $D(A)$, that is $build\_hash\_table(A) = D(A)$. get takes an as input a hash table $D(A)$ and an id x and returns a tuple from A with id x. Formally, $get(D(A), x) = a$ where $a \in A \wedge a.id = x$.

Execution Plan: We define an execution plan to be a directed acyclic graph $G = (V, E)$. Each node $v \in V$ is one of the following operations:

$\underline{probe(P, A, B, I)}$**:** $probe$ executes a predicate P over A and B using index I, and returns a set of candidate pairs $(a.id, b) \in A.id \times B$ that satisfy P. Formally, $probe(P, A, B, I) = \{(a.id, b) | (a, b) \in A \times B \wedge P(a, b)\}$.

$\underline{apply(P, D, C)}$**:** $apply$ computes a predicate on a set of input candidate pairs $C \subseteq A.id \times B$ using data structure D and returns the subset of C that satisfies P. Here D is a hash table of preprocessed tuples from A which are used to compute the comparison function of P. Formally, $apply(P, D, C) = \{(a.id, b) | (a.id, b) \in C \wedge P(a, b)\}$.

$\underline{union(C_1, C_2, \ldots, C_n)}$**:** $union$ takes two or more sets of candidate pairs $C_1, C_2, \ldots, C_n$ and returns the union of the sets. Formally, $union(C_1, C_2, \ldots., C_n) = \bigcup_{i=1}^{n} C_i$.

$\underline{intersect(C_1, C_2, \ldots, C_n)}$**:** $intersect$ takes two or more sets of candidate pairs $C_1, C_2, \ldots, C_n$ and returns the intersection of the sets. Formally, $intersect(C_1, C_2, \ldots., C_n) = \bigcap_{i=1}^{n} C_i$.

$\underline{minus(C_1, C_2)}$**:** $minus$ takes two sets of candidate pairs C_1 and C_2 and returns C_1 subtract C_2. Formally, $minus(C_1, C_2) = C_1 \setminus C_2$.

In the graph of the execution plan, each edge $(u, v) \in E$ denotes a dependency between two nodes. For example, if there is a directed edge between two nodes (u, v) then v depends on u (i.e., u must be executed before v).

For brevity, in our execution plan figures, we represent $probe$ operations with diamond shapes, $apply$ operations with rectangles, and set operations ($union, intersect$, and $minus$) with circles.

We define the semantics of an operation v in an execution plan $G = (V, E)$ as follows. Let U_v be all the input nodes to operation $v \in V$, formally, $U_v = \{u|(u, v) \in E\}$. Let $u_v^i \in U_v$ be the ith input to operation v, and P_v be the predicate associated with v. Let C_v be the output of executing operation v. For each operation type we define C_v as follows:

- *probe*: $C_v = \{(a, b)|(a, b) \in A \times B \wedge P_v(a, b)\}$
- *apply*: $C_v = \{(a, b)|(a, b) \in C_{u_v^1} \wedge P_v(a, b)\}$
- $intersect$: $C_v = \bigcap_{u \in U_v} C_u$
- $union$: $C_v = \bigcup_{u \in U_v} C_u$
- $minus$: $C_v = C_{u_v^1} \setminus C_{u_v^2}$

We define the semantics of an execution plan G to be the output of the sink node in G. That is, we recursively expand each node util we are left with a single set expression. For example,

consider the execution plan below, which we represent graphically in Figure 3.1.

$$G = (V, E)$$
$$V = \{v_1, v_2, v_3, v_4\}$$
$$E = \{(v_1, v_3), (v_2, v_3), (v_2, v_5), (v_3, v_4)\}$$

$$v_1 = probe(P_1, A, B, I) \qquad v_2 = probe(P_2, A, B, I)$$
$$v_3 = intersect(C_{v_1}, C_{v_2}) \qquad v_4 = apply(P_3, D, C_{v_3})$$

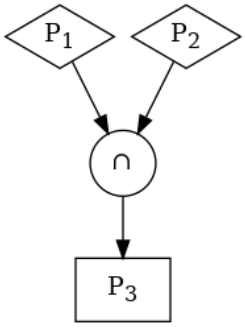

Figure 3.1: A simple execution plan.

Let C be the candidate set output by executing plan G over tables A and B. We can expand this plan into a single set expression starting at the sink node v_4.

$$
\begin{aligned}
C &= C_{v_4} \\
&= \{(a,b) | (a,b) \in C_{v_3} \wedge P_3(a,b)\} \\
&= \{(a,b) | (a,b) \in (C_{v_1} \cap C_{v_2}) \wedge P_3(a,b)\} \\
&= \{(a,b) | (a,b) \in (\\
&\qquad \{(a,b) | (a,b) \in A \times B \wedge P_1(a,b)\} \\
&\qquad \cap \{(a,b) | (a,b) \in A \times B \wedge P_2(a,b)\} \\
&) \wedge P_3(a,b)\}
\end{aligned}
$$

3.2.2 Preprocessing a Blocking Program

Given a Delex blocking program, to translate it into an execution plan, we first perform a preprocessing step. Specifically, we process the drop rules and keep rules to remove any predicates or rules that do not affect the output of the plan. In other words, we remove "redundant" predicates and rules.

The goal of preprocessing is to reduce the plan search space that we explore in later steps, by reducing the number of predicates and rules in the blocking program. We perform preprocessing in two steps, intra-rule preprocessing and inter-rule preprocessing.

Before discussing these two steps, we define several notions.

Predicate Containment: Let P_i and P_j be predicates. We say that P_i "contains" P_j, written as $P_i \sqsupseteq P_j$, if and only if the set of pairs in $A \times B$ that satisfy P_i is a superset of the pairs in $A \times B$ that satisfy P_j. Formally:

$$
\begin{aligned}
P_i \sqsupseteq P_j \iff & \{(a,b) \mid P_i(a,b) \ \wedge \ (a,b) \in A \times B\} \\
& \supseteq \{(a,b) \mid P_j(a,b) \ \wedge \ (a,b) \in A \times B\}
\end{aligned}
$$

To determine if P_i contains P_j, we first check that P_i and P_j use the same similarity function. If the similarity functions match, then we examine the comparison operation and values to determine if the $P_j(a,b) \implies P_i(a,b)$.

For example, $jaccard\_name(a,b) > .5 \sqsupseteq jaccard\_name(a,b) > .7$ because both predicates use the same similarity function $jaccard\_name$ and any pair a, b with $jaccard\_name(a,b)$ greater than .7 also has a Jaccard score greater than .5.

Rule Containment: Let R_i and R_j be conjunctions of predicates (i.e., rules). We say that R_i "contains" R_j, written as $R_i \sqsupseteq R_j$, if and only if the set of pairs in $A \times B$ that satisfy R_i is a superset of the pairs in $A \times B$ that satisfy R_j. Formally:

$$\begin{aligned} R_i \sqsupseteq R_j \iff & \{(a,b) \mid R_i(a,b) \ \wedge \ (a,b) \in A \times B\} \\ & \supseteq \{(a,b) \mid R_j(a,b) \ \wedge \ (a,b) \in A \times B\} \end{aligned}$$

We determine if $R_i \sqsupseteq R_j$ by checking that each predicate $P \in R_i$ contains at least one predicate in R_j. Formally, we can write this as

$$R_i \sqsupseteq R_j \iff \forall P \in R_i, \exists P' \in R_j, P \sqsupseteq P'$$

Additionally, we say a predicate P_j "contains" rule R_i, written as $P_j \sqsupseteq R_i$, if the rule $R_k = \{P_j\}$ contains R_i. Formally:

$$P_j \sqsupseteq R_i \iff \{P_j\} \sqsupseteq R_i$$

In Algorithm 3.1 we give the pseudocode for how we determine if one rule contains another.

Intra-Rule Preprocessing: In the first step of preprocessing we remove redundant predicates from within each rule, for both keep and drop rules. We observe that because each rule is a conjunction, if a predicate P_i contains another predicate in the rule P_j, then removing P_i leaves us with an equivalent rule. For example,

$$P_1 \sqsupseteq P_2 \implies P_1 \wedge P_2 \wedge P_3 = P_2 \wedge P_3$$

Formally, we update each rule R to be,

$$R' = R \setminus \{P_i \in R \mid \exists P_j \in R, P_i \sqsupseteq P_j \wedge i \neq j\}.$$

Algorithm 3.1 Pseudo code for determining rule containment

```
procedure RULECONTAINS(R_i, R_j)
    for P ∈ R_i do
        c ← false
        for P′ ∈ R_j do
            if P ⊒ P′ then
                c ← true
                break
            end if
        end for
        if ¬c then
            return false
        end if
    end for
    return true
end procedure
```

Inter-Rule Preprocessing: In the second step of preprocessing we remove redundant rules from the plan. Because the keep rules and drop rules are both disjunctions, if a rule R_i contains another rule R_j, then we can remove R_j without changing the output of the plan.

For example, if the keep rules are $\{R_1, R_2, R_3\}$, the drop rules are $\{R_4, R_5\}$, $R_1 \sqsupseteq R_2$, and $R_4 \sqsupseteq R_5$ then we can remove R_2 from the keep rules, and R_5 from the drop rules, leaving us with $\{R_1, R_3\}$ and $\{R_4\}$.

Algorithm 3.2 describes the pseudo code for intra-rule and inter-rule processing.

3.2.3 Generating Default Execution Plan

Once we have preprocessed a blocking program Q, we can translate it into a default execution plan G. To explain, suppose we have a preprocessed blocking program, Q, with three keep rules $\{R_1, R_2, R_3\}$ and a single drop rule $\{R_4\}$.

$$R_1 = \{P_1, P_2, P_3\} \quad R_2 = \{P_2, P_3, P_4\} \quad R_3 = \{P_5, P_1\}$$
$$R_4 = \{P_6, P_7\}$$

We construct a default execution plan as follows. First, for each keep rule $R_i \in S$, we generate a plan fragment G_{R_i} by taking the first indexable predicate P in R_i, setting P as the single source node and then apply the remaining predicates in $R_i \setminus \{P\}$ in a pipelined fashion. If the rule does not contain an indexable predicate we return an error since executing the plan would require enumerating the cross product of the input tables (which is highly impractical in most real-world settings).

After we have generated plan fragments $G_{R_1}, \ldots, G_{R_{|S|}}$, we union the output of all the plan fragments to create a single plan G_K. Next, we add the drop rules T from the blocking program by applying each drop rule on the output of G_K taking the union to get the pairs output by G_K that satisfy at least one drop rule (i.e., the pairs that must be dropped). Finally, we subtract the output the drop rules applied on the output of G_K from the output of G_K to get all pairs that match at least

Input : Rule R

Output : An equivalent rule to R with redundant predicates removed

procedure INTRARULEPREPROCESS(R)
  $X \leftarrow \emptyset$
  for $P \in R$ **do**
    for $P' \in R$ **do**
      if $P \neq P' \wedge P \sqsupseteq P'$ **then**
        $X \leftarrow X \cup \{P\}$
        break
      end if
    end for
  end for
  return $R \setminus X$
end procedure

Input : Set of rules U

Output : An equivalent set of rules to U with redundant rules removed

procedure INTERRULEPREPROCESS(U)
  $X \leftarrow \emptyset$
  for $R \in U$ **do**
    for $R' \in U$ **do**
      if $R \sqsupseteq R'$ **then**
        $X \leftarrow X \cup \{R\}$
      end if
    end for
  end for
  return $U \setminus X$
end procedure

Algorithm 3.2 Pseudo code for preprocessing.

Input : Set of rules U

Output : An equivalent set of rules to U with redundant predicates and rules removed

```
procedure RULESETPREPROCESS(U)
    U' ← ∅
    for R ∈ U do
        U' ← U' ∪ {INTRARULEPREPROCESS(R)}
    end for
    U'' ← INTERRULEPREPROCESS(U')
    return U''
end procedure
procedure BLOCKINGPROGRAMPREPROCESSING(Q)
    S' ← RULESETPREPROCESS(Q.keep_rules)
    T' ← RULESETPREPROCESS(Q.drop_rules)
    return (S', T')
end procedure
```

one of the keep rules and none of the drop rules. Continuing our example, when we translate Q we produce the default plan in Figure 3.2.

Here we note that this is not the only way that we could construct a default plan. We choose this particular method for two reasons. First, typically predicate indexing and probing are the most expensive operations that we do for a given plan execution. This default plan construction leads to the minimum number of predicates indexed without having to consider any inter-rule relationships. Second, this method imposes the fewest restrictions on the input blocking program while still allowing us to generate a plan that does not require enumerating the cross product of the input tables. Specifically, if the input blocking program has at least one indexable predicate per keep rule, we can generate an execution plan for it.

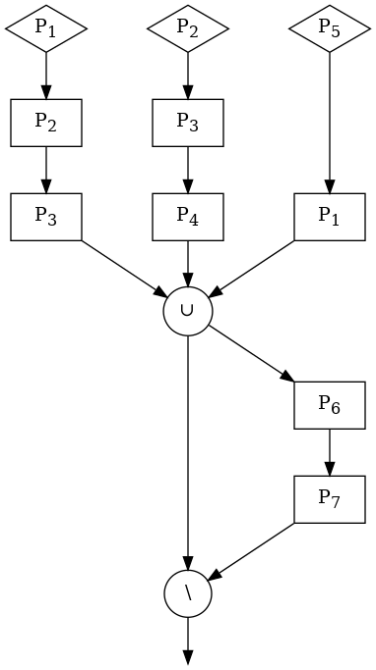

Figure 3.2: The default plan generated from the blocking program Q

3.3 Optimizing Blocking Programs

We now describe how we optimize a default execution plan. First, we discuss using rewrite rules to transform such a plan into equivalent plans. Then we discuss our cost estimation procedure used to compare possible execution plans. Finally, we discuss our optimization procedure.

In what follows, given an execution plan $G = (V, E)$, we define a "basic path" to be a directed path in G such that the first node has in-degree less than or equal to 1 and out-degree 1, and all other nodes in the path have both in-degree equal to 1 and out-degree equal to 1.

3.3.1 Rewrite Rules and Plan Space

Once we have generated a default execution plan, we can apply rewrite rules to reorder and combine operations in the execution plan to generate multiple equivalent plan variations. We consider four rewrite rules: index selection, predicate reuse, short-circuiting, and predicate reordering.

3.3.1.1 Index Selection

One of the key decisions that affect the runtime of an execution plan is which predicates we index. In the default plan, we simply choose the first indexable predicate in each keep rule. We improve upon this baseline by modeling the problem of deciding which predicates to index as a minimum weighted hitting set problem. The input of this optimization procedure is a set of keep rules, a default plan G, and a cost estimator θ (see Section 3.3.2). The output is a set of predicates to index J and a modified execution plan G'.

Minimum Weighted Hitting Set Problem: We begin with the general definition of a minimum weighted hitting set problem. Let $\{S_1, \cdots, S_n\}$ be a set of non-empty sets of elements and $U = S_1 \cup \ldots \cup S_n$.

Let $\phi : U \rightarrow \mathbb{R}$ be a cost function which takes as input an element from U and returns a real number. We define a "hitting set" X to be a subset of U where the intersection of X with each input set $S_1, \cdots, S_n$, is non-empty. Formally, $X \subseteq U, \forall i \in \{1, \ldots, n\}, |X \cap S_i| \geq 1$. A minimum weighted hitting set problem is to find a hitting set X that minimizes the sum of the costs of the

elements in X, $\sum_{x \in X} \phi(x)$. Formally,

$$\min_{X} \quad \sum_{x \in X} \phi(x)$$

$$\text{subject to}$$

$$|X \cap S_i| \geq 1, \ \forall i \in \{1, \ldots, n\}$$

$$X \subseteq U$$

For example, suppose we have three sets of integers $\{\{1, 3, 4\}, \{1, 4\}, \{2, 4\}\}$, universe $U = \{1, 2, 3, 4\}$, and cost function $\phi(x) = x^2$

$$\min_{X} \quad \sum_{x \in X} x^2$$

$$\text{subject to}$$

$$|X \cap \{1, 3, 4\}| \geq 1$$

$$|X \cap \{1, 4\}| \geq 1$$

$$|X \cap \{2, 4\}| \geq 1$$

$$X \subseteq \{1, 2, 3, 4\}$$

For this example the optimal solution $X^* = \{1, 2\}$ with a cost of 5. Typically, a minimum weighted hitting set problem is solved by creating an equivalent integer linear program (ILP):

$$\min_x \quad c^T x$$

subject to

$$Ax \geq 1$$

$$x \in \{0, 1\}^m$$

$$U = \{u_1, \ldots, u_m\} \quad A \in \mathbb{Z}^{n \times m} \quad c \in \mathbb{R}^m \quad x \in \mathbb{Z}^m$$

$$A_{ij} = \begin{cases} 1 \text{ if } u_j \in S_i \\ 0 \text{ otherwise} \end{cases} \qquad c_j = \phi(u_j) \quad x_j = \begin{cases} 1 \text{ if } u_j \in X \\ 0 \text{ otherwise} \end{cases}$$

Here we note that for a blocking program to be executed without having to enumerate $A \times B$, we must index at least one predicate which contains each of the keep rules. Additionally, because index probing is typically the most expensive operation in an execution plan, we wish to minimize the sum of the costs of the predicates that we index.

We can map this problem to a hitting set problem by mapping each keep rule R_i to an input set S_i, and redefining the cost function. Specifically, let $\{R_1, \ldots, R_n\}$ be our set of keep rules and $U = R_1 \cup \ldots \cup R_n$. Each keep rule R_i is mapped to an input set $S_i = \{P | P \in U \wedge P.indexable = true \wedge P \sqsupseteq R_i\}$. Our cost function ϕ is the cost of index build and probing for the predicate $\phi(P) = \theta_{build}(P) + \theta_{probe}(P) \cdot |B|$.

We map our index selection problem onto the ILP formulation of the minimum weighted hitting set problem as follows. First, we create a set U by taking the union of all the keep rules in Q. Next we construct a matrix $A \in \mathbb{Z}^{|Q.keep\_rules| \times |U|}$. Next, for each element $A_{ij} \in A$, if predicate $P_j \in U$ contains keep rule $R_i \in Q.keep\_rules$ and P_j is indexable, we set $A_{ij} = 1$, otherwise we set $A_{ij} = 0$. Next, we construct cost vector $c \in \mathbb{R}^{|U|}$ by setting each element $c_j = \theta_{build}(P_j) + \theta_{probe}(P_j) \cdot |B|$. We then solve the ILP using an off-the-shelf solver to get an optimal solution x^*. Finally, we construct our set of join predicates J, by iterating over x^* and adding each predicate $P_j \in U$ to J

for which $x_j = 1$. Formally, we define the variables as follows:

$$A_{ij} = \begin{cases} 1 \text{ if } P_j \sqsupseteq R_i \wedge P_j.indexable = true \\ 0 \text{ otherwise} \end{cases} \qquad x_j = \begin{cases} 1 \text{ if } P_j \text{ is indexed} \\ 0 \text{ otherwise} \end{cases}$$

$$c_j = \theta_{build}(P_j) + \theta_{probe}(P_j) \cdot |B|$$

Once we have J we can then modify the default plan to use the predicates in J as the source nodes. Specifically, for each keep rule R_i, we construct a set of all the indexed predicates that cover R_i, $R_i^J = \{P | P \in J \wedge P \sqsupseteq R_i\}$. Next, if $|R_i^J| = 1$, then we simply set the source of the path for R_i to be the single predicate and apply the remaining predicates in $R_i \setminus R_i^J$ on the output. If $|R_i^J| > 1$, then we intersect all the predicates in R_i^J and apply the remaining predicates in $R_i \setminus R_i^J$ on the output.

Continuing our example from the default plan generation. Suppose our optimal solution is $\{P_1, P_2\}$ and $P_4 \sqsupseteq R_3$. Our variables from the optimal solution to the ILP above are:

$$A = \begin{array}{c} \\ R_1 \\ R_2 \\ R_3 \end{array} \begin{array}{c} \begin{array}{ccccc} P_1 & P_2 & P_3 & P_4 & P_5 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \end{bmatrix} \end{array} \qquad c = \begin{array}{c} \\ P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \end{array} \begin{bmatrix} 50 \\ 50 \\ 100 \\ 150 \\ 100 \end{bmatrix} \qquad x^* = \begin{array}{c} \\ P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \end{array} \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Note that $A_{3\,4} = 1$ because $P_4 \sqsupseteq R_3$ even though $P_4 \notin R_3$. Once we have the optimal solution of $J = \{P_1, P_2\}$, we then transform the default plan into the plan in Figure 3.3.

3.3.1.2 Predicate Reuse

This rule takes advantage of the fact that when a union or intersection operation has two incoming basic paths which contain the same predicate P_i, we can push down the set operation and place P_i immediately after, to reduce the number of pairs that we evaluate P_i on.

For example, if we have two basic paths $P_1 \rightarrow P_3 \rightarrow P_5$ and $P_2 \rightarrow P_4 \rightarrow P_3$ that are inputs to the same union node, we can remove P_3 from both paths and apply P_3 after we have taken the union of the paths (Figure 3.4).

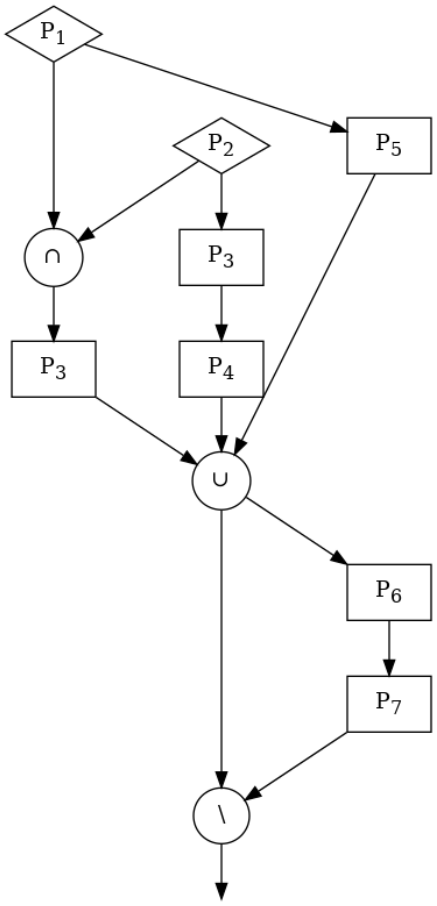

Figure 3.3: Default plan with index optimization applied.

3.3.1.3 Short Circuiting

This rule takes advantage of the fact that when we union the output of two paths, we can "short circuit" the evaluation by computing the output of the first path and then only apply the second path on the pairs that are not output by the first.

For example, if we have two rules $R_1 = \{P_1, P_2\}, R_2 = \{P_1, P_3\}$, we can join A and B on P_1, apply P_3 on the output of the join, and then apply P_2 to only the pairs that were dropped by applying P_3 (Figure 3.5).

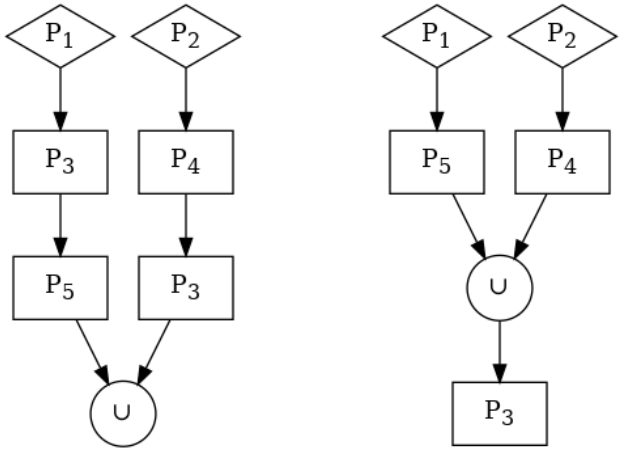

Figure 3.4: Predicate reuse example.

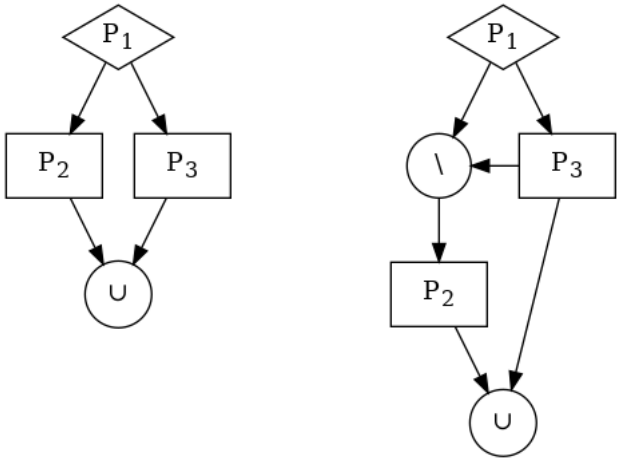

Figure 3.5: Short circuiting example.

3.3.1.4 Predicate Reordering

We observe that predicates can vary greatly in terms of their selectivity and cost of execution. To take advantage of this, we generate plan variations by reordering predicates within basic paths. For example, if we have a basic path $P_1 \rightarrow P_2$ in the execution graph G and the cost of executing P_2 is less than P_1 we can reorder the predicates to $P_2 \rightarrow P_1$ to potentially reduce the cost of executing the graph G (Figure 3.6).

We note that since we are only reordering predicates within a basic path, our predicate reorder rule does not affect the runtime of nodes outside the basic path since reordering does not affect its

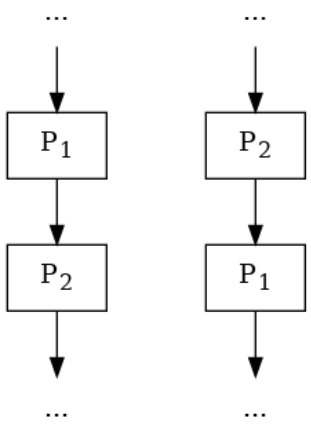

Figure 3.6: Predicate Reordering Example

output. The up shot of this is that a locally optimal reordering is also a globally optimal reordering. In practice, we do reordering by extracting all the basic paths from the execution plan G, which we call $G_1, \ldots, G_r$. Next, for each basic path G_i we estimate the cost of all permutations of G_i and use the ordering with the lowest estimated cost using our cost estimator θ (see Section 3.3.2).

3.3.1.5 Plan Space

With our rewrite rules, we can now define a plan space for an execution plan G to be $\mathcal{S}(G)$, the set of all execution plans produced by applying a sequence of zero or more rewrite rules to G. In theory, we could produce this set by iteratively applying rewrite rules to a set of plans until we reach a fixed point.

3.3.2 Cost Estimation

In order to compare plans we create a cost estimator θ, by estimating index build time, index probe time, selectivity, and application time for the predicates in Q and set operations. Once we have estimated all the necessary parameters, we then use θ to estimate the costs of execution plans. Let $\mathcal{P}$ be all the unique predicates in Q after preprocessing. Let $\mathcal{P}^I$ be all the unique indexable predicates in the keep rules of Q, i.e., all the predicates that we consider building indexes for.

Index Build and Probe Time: In order to estimate index build and probe times, we build multiple indexes on samples of A and probe each index with a sample of B. Let $\Lambda \subset \mathbb{Z}^+$ be the sample sizes that we build indexes on. Λ is currently a hyperparameter set to $\{100000, 250000, 500000\}$. We begin by taking a sample of A for each sample size in Λ, giving us $|\Lambda|$ samples, $A'_1, \ldots, A'_{|\Lambda|}$. Next, we take a sample $B' \subseteq B$. The size of B' is a hyperparameter that is current set to 10000.

Once we have sampled both A and B, for each predicate $P \in \mathcal{P}^I$, we build an index on each sample $A'_1, \ldots, A'_{|\Lambda|}$, recording the build time for each index, resulting in indexes $I_P(A'_1), \ldots, I_P(A'_{|\Lambda|})$. To estimate the build time for $I_P(A)$, we fit a linear model to the build times of $I_P(A'_1), \ldots, I_P(A'_{|\Lambda|})$, and then use this model to extrapolate the build time for $I_P(A)$, which we denote as $\theta_{build}(P)$.

Next, we probe each index with B', measuring the probe times for each record in B', and take the average probe time per record. Finally, to estimate the probe time of $I_P(A)$, we fit a linear model to the average probe times we computed using $I_P(A'_1), \ldots, I_P(A'_{|\Lambda|})$ and B' and then use the fitted model to extrapolate to estimate the probe time of $I_P(A)$ for a single tuple, which we denote as $\theta_{probe}(P, A, B)$.

Selectivity: We use two different methods to estimate selectivity, one for predicates in $\mathcal{P}^I$ and one for predicates in $\mathcal{P} \setminus \mathcal{P}^I$. Let $m = \text{argmax}_{i \in \{1,\ldots,|\Lambda|\}} \lambda_i$, that is, the index of the largest sample size in Λ. For each predicate $P \in \mathcal{P}^I$, we take the output of probing index $I_P(A'_m)$, $C_P(A'_m, B')$, which we computed when we estimated build and probe times for P. We then estimate the selectivity of P to be $\theta_{sel}(P, A, B) = |C_P(A'_m, B')|/(|A'_m||B'|)$, that is, the size of the output of probing $I_P(A'_m)$ with B' divided by the size of $A'_m \times B'$.

For predicates in $\mathcal{P} \setminus \mathcal{P}^I$ (predicates that are not possibly indexed in our final execution plan), we compute selectivity by applying on a random set of pairs and estimating the selectivity. First we take a sample $A' \subseteq A$, where $|A'|$ is a hyperparameter currently set to 100000. Next, we construct a sample of pairs by pairing each record $b \in B'$ with α records from A' to create a set of pairs $\hat{C}$. Here α is a hyperparameter, currently set to 10. Next, for each predicate $P \in \mathcal{P} \setminus \mathcal{P}^I$, we apply P to $\hat{C}$ to get the set of pairs that satisfy P, $\hat{C}_P$. Finally, we estimate the selectivity of P to be $\theta_{sel}(P, A, B) = |\hat{C}_P|/|\hat{C}|$.

Application Time: In order to estimate application times (the time it takes to apply a predicate per pair), we reuse the same sample $\hat{C}$ described above and measure the time it takes to apply the predicate for each pair in $\hat{C}$. Specifically, for each predicate in $P \in \mathcal{P}$, we apply P to each pair in $\hat{C}$ and measure the time it takes to apply P to each pair to get $t_P(i), \ldots, t_P(|\hat{C}|)$. We then estimate the time to apply P to a pair to be $\theta_{apply}(P, A, B) = \sum_{i=1}^{|\hat{C}|} t_P(i)/|\hat{C}|$.

Set Operations: For each set operation $union$, $intersect$, and $minus$ we only estimate the cost to apply each operation since there is no probe or build. To do this, we generate random sets of ids and apply the set operations. We then take the average time it takes to execute the operations to be the estimated cost of each set operation.

Plan Cost: Once we have estimated the parameters for θ we can then estimate the cost of an entire execution plan G. To do this we make one simplifying assumption that all predicates are independent and uncorrelated. That is, $\theta_{sel}(P_1 \wedge \ldots \wedge P_n) = \theta_{sel}(P_1) \times \ldots \times \theta_{sel}(P_n)$. With this assumption we can then estimate the number of pairs that each operation in our execution plan will be executed over and, consequently the runtime cost incurred by each operation. In Algorithm 3.3 we provide the pseudocode for estimating the cost of an execution plan G.

3.3.3 Searching the Plan Space

We now describe our procedure for searching the plan space to find a good execution plan. Ideally, we would do exhaustive search. That is, given a plan G, we would enumerate the entire plan space $\mathcal{S}(G)$, estimate the cost of each plan, and then output the plan with the minimum estimated cost. Unfortunately, $\mathcal{S}(G)$ can grow exponentially and hence this strategy would not scale to even moderately complex execution plans. Instead, we use beam search to search $\mathcal{S}(G)$. Specifically, we generate candidates by applying rewrite rules and retain the plans at each iteration with the smallest estimated costs. Let $\beta \in \mathbb{Z}^+$ be the number of "beams" (i.e., the number of candidates retained after each iteration). Our search procedure takes as input an execution plan G, cost estimator θ, and β and outputs an optimized plan G^*. In Algorithm 3.4 we give the pseudo-code for our optimization procedure.

Input : cost estimation parameters θ, table A, table B, execution plan G, and plan node v

Output : the estimated selectivity of v

procedure SELECTIVITY(θ, A, B, G, v)

$E \leftarrow G.edges$

$V' \leftarrow \{u | (u, w) \in E \wedge w = v\}$

if $v.type = probe$ **then**

return $\theta_{sel}(v.predicate, A, B)$

else if $v.type = apply$ **then**

return $(\prod_{u \in V'} \text{SELECTIVITY}(\theta, A, B, G, u)) \cdot \theta_{sel}(v.predicate, A, B)$

else if $v.type = intersect$ **then**

return $(\prod_{u \in V'} \text{SELECTIVITY}(\theta, A, B, G, u))$

else if $v.type = union$ **then**

return $1 - (\prod_{u \in V'} (1 - \text{SELECTIVITY}(\theta, A, B, G, u)))$

else if $v.type = minus$ **then**

$l \leftarrow \text{SELECTIVITY}(\theta, A, B, G, v.left)$

$r \leftarrow \text{SELECTIVITY}(\theta, A, B, G, v.right)$

return $l \cdot (1 - r)$

end if

end procedure

Algorithm 3.3 Pseudo code for estimating the cost of an execution plan G

Input : cost estimation parameters θ, table A, table B, and execution plan G

Output : the estimated plan cost for execution plan G

procedure ESTIMATEPLANCOST(θ, A, B, G)

$c \leftarrow 0$

$V \leftarrow G.nodes$

for $v \in V$ **do**

$U \leftarrow \{u|(u,w) \in E \wedge w = v\}$ ▷ All input nodes to v

if $v.type = probe$ **then**

$c \leftarrow c + (|B| \cdot \theta_{probe}(v.predicate, A, B) + \theta_{build}(v.predicate, A, B))$

else if $v.type = apply$ **then**

$s \leftarrow \sum_{u \in U}$ SELECTIVITY(θ, A, B, G, u)

$c \leftarrow c + (s \cdot |B \times A| \cdot \theta_{apply}(v.predicate, A, B))$

else if $v.type = union$ **then**

$s \leftarrow \sum_{u \in U}$ SELECTIVITY(θ, A, B, G, u)

$c \leftarrow c + (s \cdot |B \times A| \cdot \theta_{apply}(union, A, B))$

else if $v.type = intersect$ **then**

$s \leftarrow \sum_{u \in U}$ SELECTIVITY(θ, A, B, G, u)

$c \leftarrow c + (s \cdot |B \times A| \cdot \theta_{apply}(intersect, A, B))$

else if $v.type = minus$ **then**

$s \leftarrow \sum_{u \in U}$ SELECTIVITY(θ, A, B, G, u)

$c \leftarrow c + (s \cdot |B \times A| \cdot \theta_{apply}(minus, A, B))$

end if

end for

return c

end procedure

Input : execution plan G

Output : a set of new execution plans

procedure GENERATENEWPLANS(G)

$H' \leftarrow$ ApplyPredicateReuse(G)

$H' \leftarrow H' \cup$ ApplyShortCircuiting(G)

$H' \leftarrow \{$ApplyPredicateReordering$(h)|h \in H\}$

return H'

end procedure

Input : a set of execution plans H, cost estimation parameters θ, and number of beams β

Output : a new set of at most β execution plans

procedure TOPKPLANS(H, θ, β)

if $|H| \leq \beta$ **then**

return H

end if

$H' \leftarrow \emptyset$

while $|H'| < \beta$ **do**

$H' \leftarrow H' \cup \{\arg\min_{h \in H \setminus H'}$ ESTIMATEPLANCOST$(\theta, h)\}$

end while

return H'

end procedure

Algorithm 3.4 Plan optimization algorithm.

Input : execution plan G, cost estimation parameters θ, and number of beams β

Output : an optimized execution plan

```
procedure SEARCHPLANSPACE(G, θ, β)
    H ← {G, SELECTINDEXESWITHSETCOVER(G, θ)}
    ĉ ← ∞                                          ▷ The minimum competitive score
    while ĉ > max_{h∈H} ESTIMATEPLANCOST(θ, h) do
        ĉ ← max_{h∈H} ESTIMATEPLANCOST(θ, h)
        H' ← ∅
        for h ∈ H do
            H' ← H' ∪ GENERATENEWPLANS(h)
        end for
        H ← TOPKPLANS(H ∪ H', θ, β)
    end while
    return arg min_{h∈H} ESTIMATEPLANCOST(θ, h)
end procedure
```

3.4 Executing the Optimal Plan

In this section we discuss how we execute a blocking program after we have turned it into an execution plan.

We will consider executing the execution plan G on a Spark cluster. Let $W = \{W_1, \ldots, W_n\}$ be the worker nodes in the Spark cluster with a single driver node, Z. We assume that each worker node has the same amount of RAM M and number of threads N_t. The execution will proceed in two phases: build and execution.

The Build Phase: In the build phase we take an execution plan G and table A and output a set of data structures $\mathcal{I}$, which we will use to execute G.

First we initialize $\mathcal{I}$ to an empty directory on the disk of the driver node, Z. Next, we iterate through the operations $v \in V$ in the topological sort order. For each operation v we check if data structure $D_v(A) \in \mathcal{I}$. If $D_v(A) \in \mathcal{I}$ (i.e., $D_v(A)$ has already been built) then we simply reuse the data structure, and skip building for v. If $D_v(A) \notin \mathcal{I}$ then we call $BuildNode(v, A)$ to get $D_v(A)$, which we then write as a file on Z and add to $\mathcal{I}$.

The type of data structure D_v depends on the operation type of v. Specifically, for each operation type we build the following data structures:

- $\underline{probe(P, A, B, I)}$**:** For $probe$ operations, we build an index I_P over table A. The exact type of index depends on the comparison function of predicate P. In particular, for $jaccard$ and $cosine$ we build a prefix index. For $exact\_match$ we build a hash index which maps each unique value in the column to a list of ids from A. For all other predicates we do not implement indexes.

- $\underline{apply(P, D, C)}$**:** For $apply$ operations we simply need to obtain the preprocessed tuples from table A and store them in a hash table so that we can use them to compute the comparison function of P. In particular, for $jaccard$, $cosine$ and $overlap\_coeff$, we preprocess A by tokenizing the string column used by the comparison function in table A. For $edit\_distance$,

$jaro$, $jaro\_winkler$, and $smith\_waterman$ we preprocessed table A by simply projecting the string column used to compute the comparison function.

- *Set Operations*: For $union(C_1, C_2, \ldots, C_n)$, $intersect(C_1, C_2, \ldots, C_n)$, and $minus(C_1, C_2)$ there is no additional data required for execution beyond the sets that they operate on, hence we skip these operations in the build phase.

The Execution Phase: In the execution phase (Algorithm 3.5), we take an execution plan G, data structures $\mathcal{I}$, and tables A and B, a partition size p, and output a candidate set $C \subseteq A \times B$.

Currently, the partition size p is set to 256. We begin by distributing a copy of $\mathcal{I}$ from Z to each worker node in W. Each worker node then stores $\mathcal{I}$ on its local disk. Next, we determine the order to execute the operations in G, by performing a topological sort on G, which produces an ordered list of operations O, such that each operation in O is ordered after the operations which it depends on. Next, we split B into partitions of at most p records, $\{B_1, \ldots, B_m\}$. We then assign each partition B_i to a worker W_j, which in turn assigns B_i to a thread t. t then reads B_i into RAM. Next, t executes G over B_i in the order specified by O, using the local copy of $\mathcal{I}$ stored on the disk of W_j to produce an output set of candidate pairs C_i. Finally, we union the individual outputs $\{C_1, \ldots, C_m\}$ into a single candidate set C and return C.

Algorithm 3.5 Pseudo code executing a plan

Input : cluster of worker nodes W, table A, table B, execution plan G, and partition size p

Output : a candidate set C

```
procedure EXECUTEPLAN(W, A, B, G, p)
    V ← G.nodes
    E ← G.edges
    O ← TopologicalSort(V, E)
    C ← EXECUTENODES(W, A, B, O, E, p)
    return C
end procedure
```

Input : cluster of worker nodes W, table A, table B, topologically sorted sequence of execution nodes O, set of edges E, and partition size p

Output : a candidate set C

procedure EXECUTENODES(W, A, B, O, E, p)

$\mathcal{I} \leftarrow \emptyset$

for $v \in O$ **do**

if $D_v \notin \mathcal{I}$ **then**

$\mathcal{I} \leftarrow \mathcal{I} \cup \{\text{BUILDNODE}(v, A)\}$

end if

end for

DISTRIBUTETOWORKERS($\mathcal{I}, W$)

$C \leftarrow \emptyset$

for $B_i \in$ PARTITION(B, p) **do**

$W_j \leftarrow$ ASSIGNWORKER(B_i)

$C_i \leftarrow$ EXECUTEPLANONPARTITION(W_j, B_i, O, E)

$C \leftarrow C \cup C_i$

end for

return C

end procedure

Input : worker node W_j, partition of records B_i, topologically sorted sequence of execution nodes O, and set of edges E

Output : the candidate set for B_i

procedure EXECUTEPLANONPARTITION(W_j, B_i, O, E)
  $t \leftarrow$ GETTHREAD(W_j)
  $\mathcal{I}' \leftarrow$ GETLOCALINDEXES(W_j)
  for $v \in O$ **do**
    $U \leftarrow \{u | (u, w) \in E \ \wedge w = v\}$
    $\Delta \leftarrow$ GETNODEOUTPUTS(W_j, B_i, U)
    $C_v \leftarrow$ EXECUTENODEONTHREAD($t, B_i, \mathcal{I}', \Delta, v$)
    SAVEOUTPUTTOWORKER(W_j, C_v)
  end for
  return C_v ▷ Return the output of the last operation in O
end procedure

Input : an execution node v, and table A

Output : a data structure to execute v

procedure BUILDNODE(v, A)
  if $v.type = probe$ **then**
    return $build(v.predicate, A)$
  else if $v.type = apply$ **then**
    $A' \leftarrow preprocess(v.predicate, A)$
    return $build\_hash\_table(A')$
  else if $v.type \in \{intersect, union, minus\}$ **then**
    return $null$
  end if
end procedure

Input : set of nodes V, set of edges E

Output : topologically sorted sequence of nodes O

procedure TOPOLOGICALSORT(V, E)

$O \leftarrow (\ )$ ▷ Sorted list of nodes, initialize as empty

$U \leftarrow V \setminus \{u | (u, v) \in E\}$ ▷ Find all sinks in the graph

if $|U| \neq 1$ **then**

return Error ▷ Return error if there is not exactly one sink

end if

$v \leftarrow U(1)$ ▷ Get the unique sink

$O, U \leftarrow$ DFS$(V, E, O, \{v\}, v)$

return O

end procedure

Input : set of nodes V, set of edges E, topologically sorted sequence of nodes O, set of visited nodes U, and node v

Output : topologically sorted sequence of nodes O, set of visited nodes U

procedure DFS(V, E, O, U, v)

$U \leftarrow U \cup \{v\}$ ▷ Mark node as visited

$N \leftarrow \{u | (u, w) \in E \wedge w = v\} \setminus U$ ▷ Get all unvisited direct predecessors

for $u \in N$ **do**

$O, U \leftarrow$ DFS(V, E, O, U, u)

end for

$O \leftarrow O \parallel (v)$

return O, U

end procedure

3.5 Chunking

When the tables are very large, we have to chunk, that is, partition them so that we can execute them efficiently. In what follows we explain this issue, then describe our solution.

The key challenge with our execution strategy as described earlier is what happens when $\mathcal{I}$ becomes very large. Let M be the amount of usable RAM on any given worker node in W. As discussed above, at the start of execution, we distribute $\mathcal{I}$ to the local disk of each worker node in W. Each worker then uses its local copy of $\mathcal{I}$ to execute G over partitions of B. During the execution of G, $\mathcal{I}$ is randomly accessed when executing operations in G.

When $|\mathcal{I}| \leq M$, we are able to exploit the file system cache such that most, if not all, of the random reads that occur during execution are in the file system cache. That is, most of the reads do not require reading from disk. In contrast, when $|\mathcal{I}| > M$, we cannot fit all of $\mathcal{I}$ in the file system cache. This causes thrashing as pages from $\mathcal{I}$ are continuously being evicted from the cache and re-read from disk, which in turn can cause significant performance degradation. To address this challenge, we develop a strategy for chunking table A and executing such that the data we are randomly accessing is likely never larger than M, even when $|\mathcal{I}| > M$.

3.5.1 Working Set Size Estimation

Working Set: We define our "working set" to be all the data that we randomly access during execution. That is, our working set is the data structures in $\mathcal{I}$. Note that we omit any intermediate results and data from B that we are operating. This is because we access this data *sequentially*, which minimizes the performance impact even when we are reading from disk, as data can effectively be prefetched since the read pattern is predictable.

Given table A and operation $v \in V$, we wish to estimate the size (in bytes) of the working set of v when executing over a subset of A, that is, $|D_v(A')|, A' \subseteq A$. In other words, for each operation, we want a function $\phi_v : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ that maps the input tuples to an estimate of the size in bytes.

We assume that the operation $v \in V$ is a black box. That is, we can build v over some subset of A and get the size of the resulting data structure. In order to handle this general case, we build v over multiple samples of A and then perform non-negative ordinary least squares regression to fit coefficients to a model.

First, we must obtain data to fit a model to by taking random samples of A and building D_v over them and recording the sizes. The exact sizes and number of the samples are configurable, we currently use the sample sizes in Λ from the cost estimation procedure (Section 3.3.2).

We then fit linear coefficients to the following model:

$$\phi_v(x) = w_3 x + w_2 \sqrt{x} + w_1 \log x + w_0$$

Here x is the number of tuples in the input table, and $w_0, \ldots, w_3$ are learnable parameters. This problem can be solved with any off-the-shelf ordinary least squares solver. Once we have completed fitting the model, we can use it to predict the size of the working set of v built over some subset of A. Note that the model we use is intended to capture the scaling behavior of a wide range of indexes, hence we include both $\sqrt{x}$ and $\log x$ terms instead of just a simple linear model (e.g. $w_3 x + w_0$).

Once we have created working set size estimation functions for each operation $v \in V$, we can estimate the total working set size of an execution plan G simply by summing the workings sets of all the operations in G, specifically:

$$\text{working set executing } G \text{ over } A = \phi(G, A) = \sum_{v \in G.nodes} \phi_v(|A|)$$

3.5.2 Determining the Number of Chunks

Let M' be the target maximum working set size during execution. Currently, $M' = \min(M - N_t \cdot 2^{30}, M \cdot .75)$, that is, either M minus 1 GiB times the number of threads on a worker or 75% of M, which ever is smaller. We target a working set less than M for two reasons. First, there will likely be some amount of error in our estimate, therefore we want to add some margin to minimize the chances of thrashing. Second, each thread allocates memory to store intermediate results, which can cause our working set to be evicted from the file system cache. Therefore, we

reserve RAM for intermediate results to decrease the likelihood that pages from our working set $\mathcal{I}$ will be evicted from the file system cache.

To determine the number of chunks of A for G, we use exponential search [4] to find the minimum number of chunks such that the working set is less than the maximum allowable. Formally, we solve the following program:

$$\min \quad n$$

$$\text{subject to}$$

$$\sum_{v \in nodes(G)} \phi_v(|A|/n) \leq M'$$

$$n \in \mathbb{Z}^+$$

3.5.3 Graph Chunking

Once we have determined the minimum number of chunks, n^*, we can then split A into chunks and execute G. First, we split A into n^* chunks $\{A_1, \ldots, A_{n^*}\}$ by hashing the id column of A. Specifically, tuple $a \in A$ gets assigned to chunk $(hash(a.id) \bmod n^*) + 1$. We then executed the chunked plan as follows. For each chunk A_i, we execute G and materialize the output C_{A_i}. We then replicate C_{A_i} three ways, so that each partition of C_{A_i} is stored on three worker nodes in W. Once we have materialized the output of G executed over each chunk of A, $C_{A_1}, \ldots, C_{A_{n^*}}$, we then union all the sets to get the final output candidate set C.

We now discuss two key design decisions in our chunked execution, partitioning of intermediate sets and materializing intermediate sets.

Partitioning Intermediate Sets: We maintain each intermediate set $C_{A_1}, \ldots, C_{A_{n^*}}$, partitioned by the tuple from table B in the candidate pair and assign each partition of B to a single worker node for all chunks. Specifically, let B_j be a partition of table B and C_{A_i,B_j} be the results of executing G over chunk A_i and partition B_j. We assign B_j to a single worker W_k, which then in turn will compute $C_{A_1,B_j}, \ldots, C_{A_{n^*},B_j}$ during execution. When we replicate an intermediate output set C_{A_i}, worker W_k sends C_{A_i,B_j} to two other workers in W in order to achieve three-way

replication. Each worker stores its local partitions of the intermediate results in RAM if there is sufficient memory, otherwise it is spilled to local disk in the form of parquet files.

By using this partitioning strategy, we ensure that for any given tuple $b \in B$, there is a single worker W_j, which contains all the candidate pairs containing b across all intermediate outputs. This allows us to compute the union of all in intermediate sets, $C_{A_1}, \ldots, C_{A_{n^*}}$, without requiring any data shuffling across worker nodes, reducing the cost of performing the union.

Materializing Intermediate Sets: We materialize each C_{A_i} in order to control the working set of each worker node. By materializing the set (as opposed to executing everything in a streaming fashion), we ensure that the working set of each worker node in W is restricted to the indexes and data structures used for a single chunk of table A. Additionally, we replicate the intermediate outputs to guard against worker failures. In particular, if a worker W_j fails while computing C_{A_i}, we only need to recompute the partitions of C_{A_i} that W_j was computing when it failed, rather than all the partitions that W_j had computed from the beginning of execution.

3.6 User Defined Comparison Functions

In order to make the system extensible we allow users to define new comparison functions which can be used to write blocking programs. Let f be a user defined comparison function and predicate $P = (f, op, val)$. In order for us to use f in blocking programs the user must provide functions such that we can perform $probe$ and/or $apply$. In particular:

Function $probe$ Implementation: For $probe$ operations we require the user to provide functions for index construction and probing, specifically, $probe(P, A, B, I)$ which returns the set $\{(a.id, b) | (a, b) \in A \times B \wedge P(a, b)\}$, and $build(P, A)$ which returns index $I_P(A)$ and can then be used in $probe(P, A, B, I)$.

Function $apply$ Implementation: For $apply$ operations we require the user to provide functions to create and lookup a hash table and compute scores, specifically, $preprocess(P, A)$ which returns table A' of preprocessed tuples that contain the data to compute f, and f the comparison function.

Input : cluster of worker nodes W, table A, table B, execution plan G, partition size $p \in \mathbb{Z}^+$, RAM size M, working set size estimator ϕ

Output : the candidate set produced by executing G over A and B

procedure EXECUTEPLANCHUNKED(W, A, B, G, p, M, ϕ)

$n^* \leftarrow$ GETNUMCHUNKS($G.nodes, A, M, \phi$)

$\mathcal{C} \leftarrow \emptyset$

for $A_i \in$ PARTITION(A, n^*) **do**

$C_{A_i} \leftarrow$ EXECUTEPLAN(W, A_i, B, G, p)

REPLICATE($W, 3, C_{A_i}$)

$\mathcal{C} \leftarrow \mathcal{C} \cup \{C_{A_i}\}$

end for

$C \leftarrow \bigcup_{C_{A_i} \in \mathcal{C}} C_{A_i}$

return C

end procedure

Input : execution nodes V, table A, RAM size M, working set size estimator ϕ

Output : number of chunks to execute such that the working set size is less than M

procedure GETNUMCHUNKS(V, A, M, ϕ)

$n \leftarrow 1$

$i \leftarrow 1$

while $\phi(|A|/n) > M$ **do**

$n \leftarrow n + i$

$i \leftarrow 2 \cdot i$

end while

$i \leftarrow \lfloor i/2 \rfloor$

while $i > 0$ **do**

if $\phi(|A|/(n - i)) \leq M$ **then**

$n \leftarrow n - i$

end if

$i \leftarrow \lfloor i/2 \rfloor$

end while

return n

end procedure

3.7 Top-K Predicates

So far we have only considered predicates which use simple comparison functions. In particular, we have only considered comparison functions for which the output only depends on the two input tuples $a \in A$ and $b \in B$. In contrast, top-k predicates (e.g., $tfidf\_topk$) not only depend on the input tuples a and b but also on the other tuples in A. This key difference creates additional challenges when performing default plan generation, optimization, and chunking. Below we give details on each of these challenges and how we solve them.

3.7.1 Default Plan Generation Challenges

Recall for default plan generation we took the first indexable predicate in each keep rule and then applied the remaining predicates. For top-k predicates we cannot efficiently perform an $apply$ operation since in order to evaluate the predicate we must compute the top-k over all of table A, regardless of other operations in the plan. This means that if we simply take the first indexable predicate of a keep rule R, we may end up indexing more predicates than necessary leading to an inefficient default plan, since we are required to index all top-k predicates. To address this we modify our default plan generation as follows.

Let $Q = (S, T)$ be the blocking program which we wish to translate into a default plan. For each keep rule $R_i \in S$ we generate a plan fragment G_{R_i} as follows. First, we scan R_i to create a set R_i' of all the predicates in R_i which *must* be indexed (e.g., top-k predicates). If $|R_i'| = 0$, then we simply proceed as before, taking the first indexable predicate P in R_i, setting P as the single source node and then applying the remaining predicates in $R_i \setminus \{P\}$ sequentially.

If $|R_i'| = 1$ then we set the single predicate $P' \in R_i'$ as the single source node and then apply the remaining predicates in $R_i \setminus \{P'\}$ sequentially.

If $|R_i'| > 1$ then we set all predicates in R_i' as source nodes, intersect the output of all the source nodes and then apply the remaining predicates in $R_i \setminus R_i'$ sequentially.

After we have generated plan fragments $G_{R_1}, \ldots, G_{R_{|S|}}$, we union the output of all the plan fragments to create a single plan G_K. Next, we add the drop rules T from the blocking program

by applying each drop rule on the output of G_K taking the union to get the pairs output by G_K that satisfy at least one drop rule (i.e., the pairs that must be dropped). Finally, we subtract the output the drop rules applied on the output of G_K from the output of G_K to get all pairs that match at least one of the keep rules and none of the drop rules.

3.7.2 Optimization Challenges

Our index selection rewrite rule assumed that we can $apply$ any given predicate in any keep rule. As mentioned above this is not possible for top-k predicates. In fact this assumption could lead us to create a more expensive plan if we index non-top-k predicates for rules which have top-k predicates since we must index the top-k predicates.

To address this challenge we modify our index selection rewrite rule as follows. Let G be an execution plan generated from blocking program $Q = (S, T)$. First, we scan all keep rules in S to create a set L of all the predicates which must be indexed. That is, $L = \{P | R \in S \wedge P \in R \wedge \neg P.has\_apply\}$.

Next, we create a new set of keep rules S' containing only the keep rules in S for which all predicates in the keep rule can be used in an $apply$ operation. Formally, $S' = \{R | R \in S \wedge |R \cap L| \geq 1\}$. Once we have created S', we select predicates to index for the keep rules in S' by modeling index selection as a minimum weighted hitting set problem as in Section 3.3.1.1. From this procedure we get a set of predicates to index J. We remove any predicates in L which are contained by other predicate in L to get a new set $L' = \{P | P \in L \wedge \nexists P' \in L, P' \sqsupseteq P \wedge P' \neq P\}$. We then union J and L' to get the final set of predicates which we will index, J^*.

Once we have J^* we can modify the plan G to use the predicates in J^* as the sources. Specifically, for each keep rule R_i, we construct a set of all the indexed predicates that cover R_i, $R_i^J = \{P | P \in J^* \wedge P \sqsupseteq R_i\}$. Next, if $|R_i^J| = 1$, then we simply set the source of the path for R_i to be the single predicate and apply the remaining predicates in $R_i \setminus R_i^J$ on the output sequentially. If $|R_i^J| > 1$, then we intersect the output of all predicates in R_i^J and apply the remaining predicates in $R_i \setminus R_i^J$ on the output sequentially.

3.7.3 Chunking Challenges

Currently, we have laid out a fairly simple method of chunking where we simply execute the plan over multiple chunks on table A and then union the results of each chunk to get a final candidate set C. Top-k predicates add two challenges with this scheme with regard to correctness and efficiency.

The first challenge of using top-k predicates with chunking is correctness. Let $Q(A, B)$ be all the pairs in $A \times B$ which satisfy blocking program Q. Let A_1, A_2 be two halves of table $A = A_1 \cup A_2$. Consider two simple blocking programs $Q_1 = (\{(jaccard(a, b), >, .7)\}, \emptyset)$ and $Q_2 = (\{(tfidf\_topk(a, b, 10), =, 1)\}, \emptyset)$. Because Q_1 only uses a simple threshold based comparison we have $Q_1(A_1, B) \cup Q_1(A_2, B) = Q_1(A, B)$, that is, we can execute Q_1 over A_1 and A_2 independently and union the results to get the same set of candidate pairs as simply executing Q_1 over A.

In contrast, $Q_2(A_1, B) \cup Q_2(A_2, B) \neq Q_2(A, B)$, therefore we cannot simply treat top-k predicates like we do other predicates. The naive solution to the above challenge of correctness is to simply track the necessary information with each candidate pair to resolve the top-k when we compute the union of $Q_2(A_1, B) \cup Q_2(A_2, B)$. This solution comes with some significant disadvantages in terms of code complexity and efficiency.

In terms of complexity, in the simplest case we must persist the output of each top-k predicate in the execution plan G for each chunk of A we execute G over, in order to resolve the true top-k over A and produce the correct candidate set. Note that more information may need to be persisted depending on how the top-k results are combined in G, which adds even more complexity.

In terms of efficiency, the naive solution has a significant cost in both space and compute efficiency. For space efficiency, as mentioned above, we must persist additional data to produce the correct results, which adds potentially large amounts of space overhead.

For compute efficiency, each top-k predicate produces at most k pairs for each tuple in B, however with the naive solution, will produce at most k pairs for each tuple in B *per chunk of table* A. This means that if we have split A into n chunks, we will do up to n times more work than if we had resolved the true top-k over A before executing the rest of the plan.

Finally, many indexes for top-k become more efficient as you index large amounts of data. For example, Hierarchical Navigable Small World indexes (HNSW indexes) have the search complexity $O(\log N)$ where N is the number of records indexed. If we divide the records into two chunks, then the complexity becomes $O(2\log(\frac{N}{2})) = O((\frac{N}{2})^2) > O(\log N)$.

In order to address the challenges of chunking with top-k predicates, we modify our chunking procedure as follows. First, we scan the plan and collect all the nodes with top-k predicates which we call U. Next, we perform a topological sort on the plan G to get a sorted list of non-top-k nodes O.

For each top-k node in $v \in U$, we execute the node individually, splitting A into chunks if necessary. Specifically, we determine the number of chunks n_v using ϕ and exponential search [4]. Once we have the number of chunks n_v, we then split A into n_v by hashing the id column of A to get $\{A_1, \ldots, A_{n_v}\}$. We then initialize the final output of node v, $C_v = \emptyset$. For each chunk A_i, we execute v to get a partial top-k set C_{v,A_i}. We then update C_v by merging the partial top-k C_{v,A_i}, after which we replicate the updated C_v three ways.

Once we have computed the complete top-k results for each node in U we proceed as before. We determine the minimum number of chunks n^* that we need to execute the nodes in O using ϕ and exponential search [4]. Next, we split A into n^* chunks $\{A_1, \ldots A_{n^*}\}$ by hashing the id column of A. That is, tuple $a \in A$ gets assigned to chunk $(hash(a.id) \bmod n^*)+1$. For each chunk A_i, we execute the nodes in O and materialize the output C_{A_i}. We then replicate C_{A_i} three ways, such that each partition of C_{A_i} exists on three worker nodes in W. Once we have materialized the output of G executed over each chunk of A, $C_{A_1}, \ldots C_{A_{n^*}}$, we then union all the sets to get the final output candidate set C.

3.8 Experiments

We now present our evaluation of Delex. We begin with evaluating the effectiveness of our optimization procedure for reducing runtime. Next, we examine the scaling behavior of delex. We then give a qualitative examination of the expressiveness of Delex. Finally, we demonstrate the effectiveness of chunking on large datasets.

Input : cluster of worker nodes W, table A, table B, execution plan G, partition size p, RAM size M, working set size estimator ϕ

Output : An equivalent set of rules to U with redundant predicates and rules removed

procedure EXECUTEPLANCHUNKED(W, A, B, G, p, M, ϕ)

$V \leftarrow G.nodes$

$E \leftarrow G.edges$

$U \leftarrow \{v | v \in V \wedge is\_topk(v) = true\}$

$O \leftarrow$ TopologicalSort$(V, E) \setminus U$ ▷ Topological sort of all non-topk nodes

for $v \in U$ **do**

$n_v \leftarrow$ GETNUMCHUNKS$(\{v\}, A, M, \phi)$

$C_v \leftarrow \emptyset$

for $A_i \in$ PARTITION(A, n_v) **do**

$C_{v,A_i} \leftarrow$ EXECUTENODES$(W, A_i, B, \{v\}, \emptyset, p)$

$C_v \leftarrow$ MERGETOPK(C_v, C_{v,A_i})

REPLICATE$(W, 3, C_v)$

end for

end for

$n^* \leftarrow$ GETNUMCHUNKS$(G.nodes, A, M, \phi)$

$\mathcal{C} \leftarrow \emptyset$

for $A_i \in$ PARTITION(A, n^*) **do**

$C_{A_i} \leftarrow$ EXECUTENODES(W, A_i, B, O, E, p)

REPLICATE$(W, 3, C_{A_i})$

$\mathcal{C} \leftarrow \mathcal{C} \cup \{C_{A_i}\}$

end for

$C \leftarrow \bigcup_{C_{A_i} \in \mathcal{C}} C_{A_i}$

return C

end procedure

All experiments were run on an AWS cluster of 10 nodes. Each node is a m5.x4large with 16 vCPU and 64 GB RAM costing $0.768/hour (as of July 2024).

Datasets: We used five datasets for evaluating Delex: FEC, FEC 2022 January, DBLP, Music Brainz, and WDC (see Table 3.1). FEC is the DIME dataset from [6] which contains political contribution information. FEC 2022 January is a subset of the FEC dataset [6], note that we use this smaller subset for most of our experiments since the full FEC dataset is 156M records and hence was not practical to run for many experiments. DBLP dataset [22] is from and contains paper citations. Music Brainz is a synthetically generated dataset from [38], where each row refers to a song. WDC is from [59] and contains product listings extracted from the Common Crawl [27].

Dataset	$\|A\|$	Number of Columns
FEC	156M	6
FEC 2022 January	5M	6
DBLP	7.2M	11
WDC	26.5M	8
Music Brainz	20M	7

Table 3.1: Datasets used for Delex evaluation.

Blocking Programs: All programs used to evaluate Delex were handwritten by an expert user. In Table 3.2 we give a summary of the blocking programs that we used to evaluate Delex, (see Appendix A for the full blocking programs).

Program	Keep Rules	Drop Rules	Has Top-k
FEC	3	1	True
FEC 2022 January Program 1	3	2	False
FEC 2022 January Program 2	3	2	True
DBLP Program 1	4	1	False
DBLP Program 2	3	1	True
WDC Program 1	3	2	False
WDC Program 2	2	0	True
Music Brainz Program 1	5	1	False
Music Brainz Program 2	4	2	True

Table 3.2: Programs used to evaluate Delex.

3.8.1 Optimization

We begin by examining the effectiveness of the optimizer by comparing the runtime of each blocking program with default plan generation and with optimized plan generation. The results are reported in Figure 3.7. Each subplot corresponds to a single dataset. Each bar shows the total runtime of a single execution plan generated for one of two programs for its respective dataset. Additionally, each bar is further divided into the runtime for individual steps of the execution, cost estimation time (light green), optimization time (teal), build time (blue), and execution time (orange).

For example, the rightmost bar in the first subplot shows the runtime for optimized program 2 on FEC 2022 January, where the total runtime was approximately 25 minutes, cost estimation took 4.6 minutes, build took 2.2 minutes, execution took 16.3 minutes and optimization ran in a few seconds (too small to be visible in the plot).

First, we observe that the optimized plans run 1.5 to 3 times faster than the default plans for all programs except FEC 2022 January Program 1, where the default plan runs faster than the optimized one. Examining the runtime of each component we see than while the optimizer did produce a plan which runs slightly faster than the default plan, the cost estimation overhead causes the overall runtime to be slower.

Second, we note that for all programs except FEC 2022 January Program 1, execution takes the majority of the total runtime, while the build time is a relatively small proportion, cost estimation takes a roughly constant amount of time for all runs (roughly 5 minutes), and optimization time takes a negligible amount of time (less than 5 secs) for all jobs

3.8.2 Scaling

We now examine the scalability of Delex as we increase the dataset size and the scaling behavior as we add more nodes to the cluster.

Dataset Size: To examine the scaling behavior of Delex with respect to dataset size we ran Delex on the various subsets of WDC and Music Brainz and plotted the runtime. For each dataset we ran

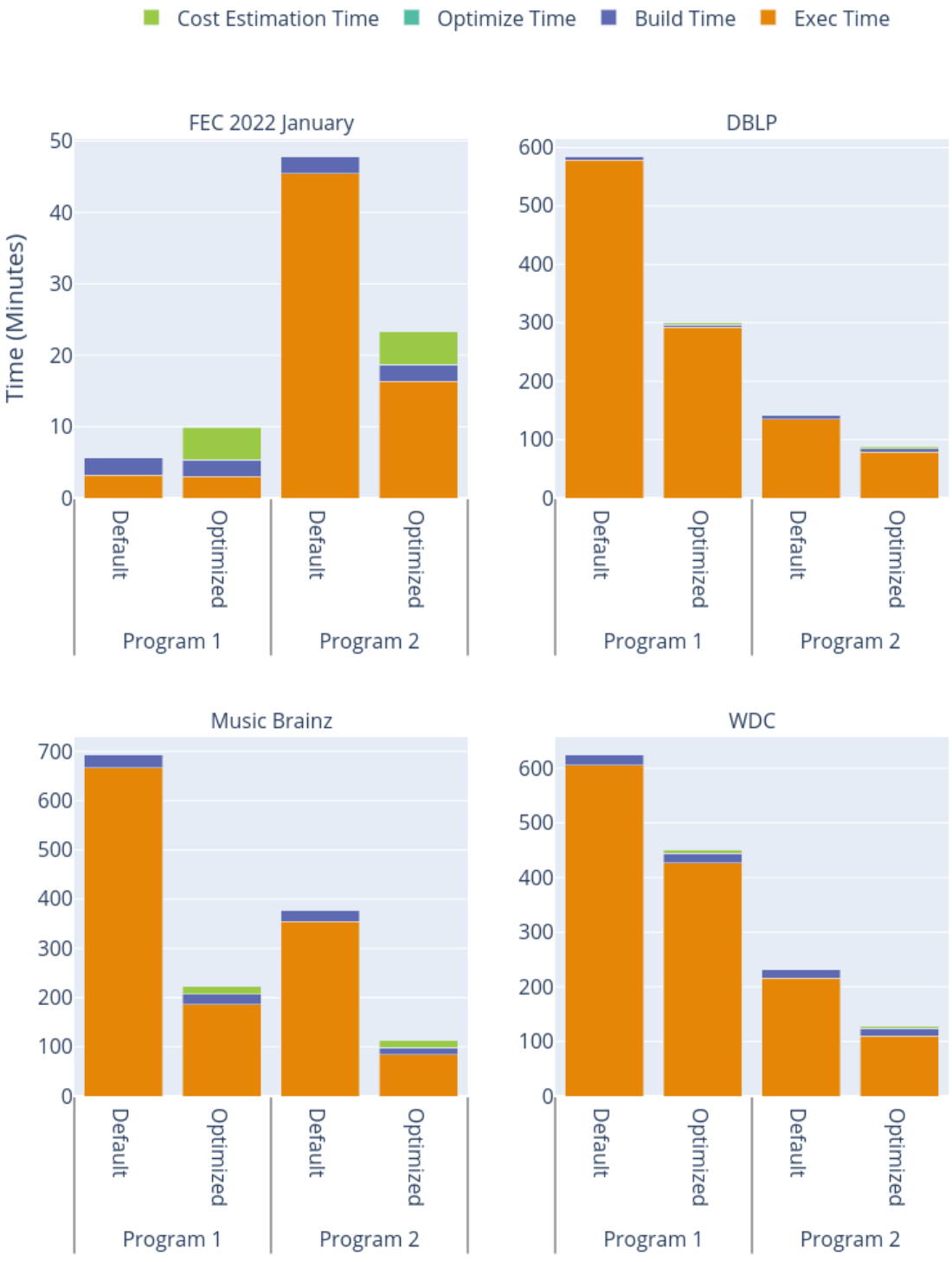

Figure 3.7: Comparison of default plan vs. optimized plan runtimes

two programs with and without optimization enabled, for the 5M, 10M, 15M, and 20M samples of the dataset. Each sample was generated by randomly sampling the datasets without replacement.

Figure 3.8 shows the total runtime for each of these runs. Specifically, the orange lines are for Program 1, the blue lines are for Program 2, and the line style indicates whether the execute plan was default (dashed line) or optimized (solid line). For example, the top subplot shows that Program 1 optimized on Music Brainz ran in 100.5 minutes for the 15M record sample of Music Brainz.

From these experiments we observe two patterns. First, Music Brainz Program 1 and WDC Program 1 scale roughly quadratically for both default and optimized plans. We note that since we do not adjust the thresholds based on the size of the input datasets and both programs do not contain top-k predicates, the output size of these programs increases quadratically. This means that these programs are inherently quadratic since the lowest runtime that can be achieved is relative to the size of the output. For such programs, while optimization does not change the overall scaling behavior, the savings from optimization becomes very significant as the datasets get larger.

The second pattern we observe is that Music Brainz Program 2 and WDC Program 2 scale near linearly with dataset size. In contrast to Music Brainz Program 1 and WDC Program 1, these programs use top-k predicates in their keep rules. By including top-k predicates, we bound the output size to be linear with the size of the dataset, and hence we make it possible to achieve sub-quadratic scaling. Additionally, the runtime of the algorithm we use to evaluate BM25 top-k predicates grows sub-linearly, allowing us to achieve an overall super-linear but sub-quadratic runtime, which is consistent with our evaluation of Sparkly from Chapter 2.

Cluster Size: Next we examine the scale-out behavior of Delex. To do this, we ran Program 1 and 2 for FEC 2022 January and DBLP, and varied the cluster size and reported the runtime in Figure 3.9. Each subplot in the figure corresponds to a single dataset (FEC 2022 January or DBLP) and the line color corresponds to the program (orange for Program 1, blue for Program 2). For example, the top subplot shows that Program 1 ran in 9.7 minutes on a cluster of 10 nodes.

Here we observe that the runtime significantly decreased by increasing the size of the cluster for all programs except FEC 2022 January Program 1. Similar to the results in Figure 3.7, FEC

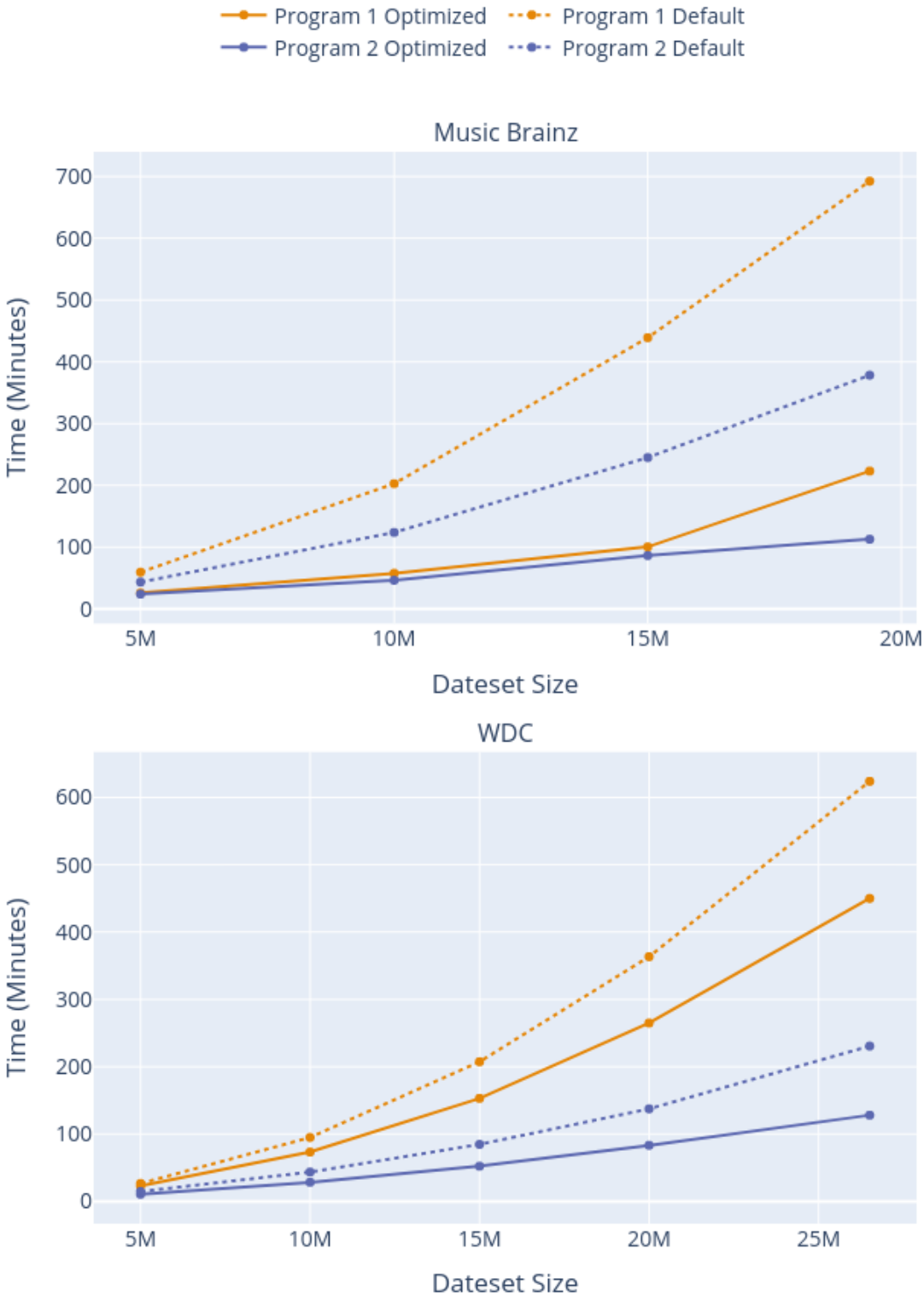

Figure 3.8: Scaling with respect to dataset size.

2022 January Program 1 runs in just a few minutes, meaning that as we increase the cluster size, eventually the runtime is dominated by overhead.

3.8.3 The Expressive Power of Delex

We now examine the expressive power of Delex by comparing it to common blocking methods in recent literature. Starting with the most basic blocking methods, hash / equality, sorted neighborhood, similarity based, and top-k blocking can all be easily expressed with a blocking program that has a single keep rule with a single predicate corresponding to the blocking method.

Additionally, Delex can be used to emulate blockers from recent papers such as Falcon [19], Corleone [28], CloudMatcher [29], DeepBlocker [62], and Sparkly [57].

One notable blocker which Delex cannot emulate is many JedAI [54] blocking workflows. JedAI blocking workflows frequently have algorithms which operate on the candidate set as a whole, where Delex is restricted to only operating on the pairs with a single probe record $b \in B$ at a time.

For example, Delex allows top-k for a single record in B whereas JedAI allows computing global top-k of the entire candidate set. This restriction in Delex is by design and exists to improve the scalability of the system. By disallowing dependencies across probe records, a single node can compute all output pairs containing a probe record $b \in B$ without requiring data from any other worker node. With this restriction we can ensure that, in the absence of worker failures, we never have to shuffle data across nodes. Data shuffling (such as a sorting the candidate set) in Spark is known to be a very expensive operation hence removing such operations improves scalability significantly.

3.8.4 Chunking

Finally, we examine the effectiveness of chunking for very large datasets by running the full FEC 2022 dataset of 156M tuples with and without chunking. For this experiment, we ran with a cluster of 10 nodes where each worker node had 16 vCPUs but half the RAM (32 GB) of the standard node. In particular, each worker node was a c5.4xlarge with 16 vCPU and 32GB RAM

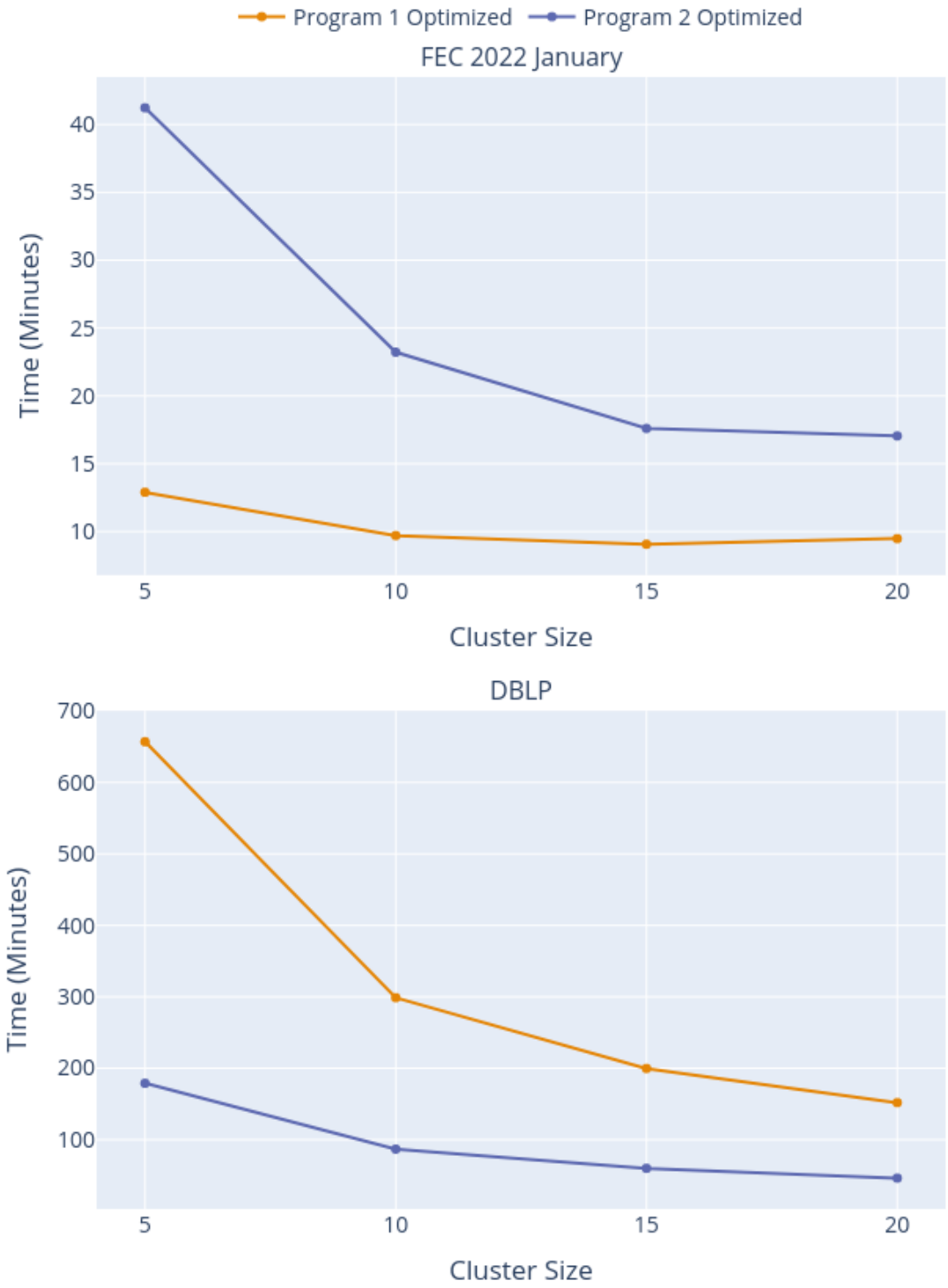

Figure 3.9: Scaling with respect to cluster size.

costing $0.68/hour (as of July 2024). For this test we kept the driver node the same as in the other experiments to ensure that the build step does not fail for the run where we disabled chunking since chunking is used to limit RAM usage during the build and the execution phase.

The results of this experiment are presented in Table 3.3. For the run with chunking disabled, we allowed the program to run for 24 hours, at which point it had reached a steady state of execution allowing us to estimate the total runtime to be approximately 8 days. In contrast, the run with chunking enabled completed end to end in 13 hours and 16 minutes. The large difference in runtime is due to the indexes no longer fitting in the memory of the worker nodes. Specifically, when no chunking was used, the total size of the indexes was approximately 32GB, which caused the worker nodes to thrash during execution and slowed execution by an order of magnitude.

	w/o Chunking	w/ chunking
Cost Estimation	10 mins	10 mins
Optimization	1 sec	1 sec
Build	6 hrs 55 mins	3 hrs 38 mins
Execution	8 days +	9 hrs 28 mins
Total	8 days +	13 hrs 16 mins

Table 3.3: Runtime on FEC with and without chunking.

3.9 Discussion and Future Work

In this work we have demonstrated the promise of Delex as a unified solution for combining blocking methods in a scalable way. We have identified three major areas that future work can build upon Delex in terms of writing blocking programs, optimization, and evaluation.

First, perhaps the most important area for future work to explore is writing blocking programs. One of the key advantages of Delex is its expressive blocking language. However, this expressiveness presents a challenge for human users to write blocking programs.

- First, writing blocking programs typically involves making small adjustments to an existing program, such as adjusting a threshold or adding / removing a predicate. Currently, testing these changes requires re-executing the entire blocking program from scratch. Future

work could build upon Delex's execution engine to cache intermediate results and avoid re-executing everything when blocking programs are changed slightly.

- Second, Delex currently does not have any built-in feedback to help the user understand the output of a blocking program. Future work could add tools (similar to [41]) to analyze the output of blocking programs to help guide the user and achieve target recall, output size, and/or runtime.

The second line of future work is that, while Delex already has an effective optimization algorithm, there are multiple ways in which future work could improve our optimization. First, we only check for predicate containment when both predicates use the same similarity function. While this covers most cases of predicate containment there are certain similarity functions which have relationships to others. For example, the overlap coefficient score between two sets is always greater than or equal to the Jaccard score between the same sets. Such information could further improve the possible plan transformations considered during optimization. Second, our optimization algorithm currently uses generic beam search to find a good plan. Future work could explore specialized search algorithms tailored to this specific use case and possibly find better plans.

Finally, future work could extend our evaluation in three distinct ways. First, Delex can be evaluated on more datasets, specifically ones that cover data domains not in the current evaluation. Second, the scalability of Delex can be further evaluated using even larger datasets than those used in our experiments. Third, the evaluation of Delex's extensibility can be expanded via a user study. For example, participants could be tasked with implementing a predicate with Delex and another existing system and compare the implementation time for each.

Chapter 4

BigGoat: A Scalable Benchmark for Blocking

In this chapter we describe BigGoat, a benchmark to evaluate how scalable a blocking technique is. We first motivate BigGoat. Second, we describe its components and how to use it. Third, we discuss downsampling, a core technique that lies at the heart of creating BigGoat. Fourth, we discuss experiments evaluating the quality of BigGoat. Finally, we discuss the need to scale up BigGoat to even bigger sizes, and our preliminary work and findings on this topic.

4.1 Motivation

As we discussed earlier in this dissertation, numerous blocking solutions for EM have been developed. These blocking solutions have mainly been evaluated with respect to *recall* and *output size*, as ideally we want a blocking solution that produces a relatively small output size, but with high recall. Here recall is defined as the fraction of true matches that appear in the blocking output.

Yet in practice, we have observed that users often struggle with scaling the blocking solutions. This is because users (especially at mid-size to large enterprises) often must match datasets consisting of 50M to 500M tuples and more. At this scale, scaling becomes a serious challenge. Many existing blocking solutions simply do not scale. They would take weeks to run, or just crash after several hours (e.g., when they run out of memory).

This problem is so serious that we have observed many users evaluate whether to use a blocking solution primarily based on whether that solution scales. If they judge the solution not scalable, then they will not use the solution in the first place. In this case, recall and the output size come in

as secondary factors: only if the users judge a blocking solution scalable would they consider its recall and output size.

Consequently, we believe that researchers must devote significantly more effort to developing blocking solutions that scale, in order to maximize the adoption and impact of these solutions in practice. Toward this goal, in this third part of the dissertation, we develop BigGoat, a scaling benchmark for blocking solutions. To our knowledge, such benchmarks have not existed before (see the related work section).

4.2 The BigGoat Benchmark

In this section we first discuss the context of how users often fine tune and use blocking solutions in practice. This context provides the guidance for designing BigGoat. Next, we describe the components of BigGoat. Finally, we describe how this benchmark should be used.

4.2.1 The Context

To design BigGoat, we consider how blocking is used in practice. We focus on the common scenario of matching two tables A and B. For this scenario, there are typically two stages. The first stage, often called *development*, consists of the following steps:

- The user decides to use a particular blocking system Q, using whichever knowledge he or she has about blocking and the two tables A and B.

- The user then evaluates to see if Q is suitable for A and B. Toward this goal, the user creates a small sample (A', B', G'), where A' is a small table sampled from A, B' is a small table sampled from B, and G' is the set of all gold (i.e., correct) matches between A' and B'. Currently, there is no guidance for how to do this, so in practice this is typically done in an ad-hoc fashion.

- The user evaluates blocker Q on (A', B', G'). A major focus here is to *tune* Q to achieve a certain recall threshold on A' and B', e.g., 90%. The user can do this because he or she has access to the gold matches G' and thus can compute the recall. Other considerations involve

how big the output size and how long blocker Q takes to run on A' and B'. In this work we will focus on recall and assume that the user has tuned blocker Q to reach a desired recall threshold on A' and B'.

For example, suppose we have two tables A, B with $|A| = |B| = 100,000,000$, and Q is Sparkly Manual. We first downsample A and B to get A' and B' with $|A'| = |B'| = 500,000$. Next, we create the set of gold labels G' for A', B'. Finally, we tune blocker Q, that is, test different blocking fields and k values, using A', B' and G' to estimate our recall, and pick the combination of blocking fields and k value based on the estimated recall.

As described, the key challenge of the development stage is how to downsample A and B such that the recall of Q on A and B is close to the recall estimated using A', B' and G'. In Section 4.4 we examine this problem in detail and give a novel solution for this problem.

In the second stage, called *production*, the user deploys the tuned blocker Q to perform blocking for the two original tables A and B. The implicit assumption here is that the tuned blocker Q will reach approximately the same recall on A and B, as the case on the sample tables A' and B'.

Depending on how the blocker was created in the development stage, deploying the tuned blocker can be as simple as adding more resources to a cluster or as complex as completely reimplementing the blocker with a new architecture. Continuing our example with Sparkly Manual, after the development stage we would have chosen columns and a k value. In the production stage, we would then add nodes to the cluster to speed up running on the full dataset.

The two key challenges in the production stage are how to execute Q over A and B both efficiently and reliably. In terms of efficiency, the challenge is how to execute over Q the full A and B with in a set time period for a given hardware setting. For reliability, Q must handle a variety of faults and exceptions, such as machine failures, network failures, disk faults, and memory errors.

4.2.2 Goals, Components, and Usage of BigGoat

We want to design the BigGoat benchmark to support the above usage scenario. In particular, we focus on supporting the deployment stage. That is, we want a benchmark that can help evaluate

whether a blocker Q scales to large tables (of the sizes commonly seen in practice today and in the foreseeable future).

Toward this goal, we envision that the BigGoat benchmark will consist of a set of datasets $D_1, D_2, \ldots, D_n$. Each of these datasets D is a quintuple (A, B, A', B', G'), where A and B are two large tables to be matched, A' is a downsample of A, B' is a downsample of B, and G' is the set of gold matches between A' and B'.

We propose to use the BigGoat benchmark as follows (see Algorithm 4.1). To evaluate a blocker Q, we should tune Q on a dataset D in the benchmark, by modifying Q to achieve a pre-specified recall (e.g., 90%), on A' and B' (we can compute this recall using G'). We suggest using the default recall levels of 80%, 90%, and 95%.

Let the tuned blocker at a particular recall level be Q'. We then execute Q' over the original tables A and B, and report the recall level, the hardware setting, the runtime, and the output size.

This way, we can evaluate how a blocker performs in a setting that closely approximates real-world settings, e.g., understanding if we can use it to block two large tables of 50M tuples each in a reasonable amount of time, using 10 AWS commodity machines. We can also compare blockers, e.g., comparing blockers Q_1 and Q_2, using a five node AWS cluster on a particular dataset, with a target recall of 90%.

4.2.3 Challenges

Developing BigGoat, as envisioned above, raises the following challenges.

- First, we must find large datasets, each of which must either have two tables A and B, or one table A (so that we can match it with itself). Each table should ideally have tens of millions to hundreds of millions of tuples, as these sizes are common today in real-world EM scenarios.

- Second, we must develop a downsampling procedure such that given two tables A and B, we can downsample them to A' and B' such that if we tune a blocker Q on the downsamples

Algorithm 4.1 Pseudocode for running the benchmark

Input : Datasets $\mathcal{D}$, recall targets $T \subset \mathbb{R}$, hardware setting H, and blocker Q

Output : Experimental results $X \subset \mathcal{D} \times T \times \{H\} \times \mathbb{R} \times \mathbb{Z}^+$

procedure RUNBIGGOAT($\mathcal{D}, R, H, Q$)
  $X \leftarrow \emptyset$
  for $(A, B, A', B', G') \in \mathcal{D}$ **do**
    for $t \in T$ **do**
      $Q' \leftarrow Q$
      while $|Q'(A', B') \cap G'|/|G'| \leq t$ **do** ▷ Wile recall on sample is less than target
        $Q' \leftarrow$ Tune(Q') ▷ User tunes Q' to increase recall on sample
      end while
      $s_1 \leftarrow$ Time()
      $C \leftarrow$ RunBlocker(Q', H, A, B) ▷ Run Q' over A and B using hardware H
      $s \leftarrow$ Time() $- s_1$
      $X \leftarrow X \cup \{((A, B, A', B', G'), t, H, s, |C|)\}$
    end for
  end for
  return X
end procedure

to reach a target recall t, then when executing Q on A and B we also reach recall t or very close to it.

In the rest of this chapter, we describe the BigGoat benchmark and how we address these two challenges.

Dataset	$\|A\|$	$\|B\|$	$\|A'\|$	$\|B'\|$	$\|G'\|$	Number of Columns
Big Citation	1.8M	2.5M	188K	245K	90K	6
FEC	60M		500K		1.1M	6
WDC	18M		500K		167K	8
Music Brainz	20M		500K		377K	7
North Carolina Voters	10M		500K		777K	4

Table 4.1: The five datasets of BigGoat.

4.3 The Datasets of BigGoat

The BigGoat benchmark consists of five datasets: Big Citation, FEC, WDC, Music Brainz, and North Carolina Voters. We now describe how we created each dataset. Table 4.1 provides statistics on the datasets.

4.3.1 Big Citation

The Big Citation dataset is from [19]. It consists of two tables of paper citations, one from CiteSeer with 1.8M records and one from DBLP with 2.5M records. The labels for this dataset were generated using hand tuned heuristic rules. For BigGoat we reformatted the dataset but otherwise leave the data untouched. Below we give two example records from the dataset.

```
{
 'title': 'A Lightweight Object Manager for Group-Aware Applications  GMD 1999 '
          'GMD ',
 'authors': 'Thomas Tesch, Marc Volz, Forschungszentrum Informationstechnik '
            'Gmbh, Schlo Birlinghoven, Gebrauch Verffentlicht Jegliche '
            'Inhaltsnderung, Thomas Tesch, Marc Volz',
 'journal': None,
 'month': None,
 'year': None,
 'publication_type': None
}
```

```
{
 ‘title’: ‘Functional Characterization of a Trypanosoma brucei TATA-Binding ’
          ‘Protein-Related Factor Points to a Universal Regulator of ’
          ‘Transcription in Trypanosomes’,
 ‘authors’: ‘Jia-peng Ruan, George K. Arhin, Elisabetta Ullu, Christian ’
            ‘Tschudi, Departments Of Epidemiology, Public Health, Internal ’
            ‘Medicine, Cell Biology’,
 ‘journal’: None,
 ‘month’: None,
 ‘year’: 2004,
 ‘publication_type’: None
}
```


4.3.2 FEC

The Federal Election Commission (FEC) dataset was created by combining two datasets, the individual contributors database published by the FEC [16] and the Stanford DIME dataset [6]. The former as of 2024 was approximately 200M records and each record was a political campaign contribution reported to the FEC. The latter is public data published by the FEC that has been cleaned and normalized by researchers at Stanford for analyzing political contributions. Our version of the FEC dataset joins the raw government dataset and the DIME dataset to create a new dataset with gold matches. Specifically, we join the individual contributors database and DIME dataset on shared ID columns, and then clean the labels produced by dropping any records with likely erroneous labels and truncating clusters to at most 50 records. Below we give two example records from the dataset.


```
{
 ‘city’: ‘SUNNYSIDE’,
 ‘employer’: ‘NOT EMPLOYED’,
 ‘name’: ‘TREDANARI, ADRIANA’,
```

```
 ‘occupation’: ‘NOT EMPLOYED’,
 ‘state’: ‘NY’,
 ‘zip_code’: ‘11104’
}
{
 ‘city’: ‘WILLIAMSTOWN’,
 ‘employer’: ‘CTX INFRASTRUCTURE’,
 ‘name’: ‘KOPLINER, CHRIS’,
 ‘occupation’: ‘OPERATOR’,
 ‘state’: ‘NJ’,
 ‘zip_code’: ‘19034’
}
```

4.3.3 WDC

The Web Data Commons (WDC) dataset is a cleaned version of the WDC dataset published by the University of Mannheim [59]. The original dataset is structured data extracted from the Common Crawl [27] dataset. The original dataset has gold labels automatically generated using extracted product ids as well as a small number of hand labeled pairs used for evaluating matching algorithms. We processed this dataset by removed trivial matches for which the product title and description are identical. Below we give two example records from the dataset.

```
{
 ‘name’: ‘4 5mm diamond flower thread stud’,
 ‘description’: ‘a charming earring with an antique feel the maria tash ’
                ‘diamond flower threaded stud features a feminine scalloped ’
                ‘edge with bold diamond detailing a dainty way to ’
                ‘differentiate from other piercings ,  affixing the concept of ’
                ‘beauty to durability and comfort maria tash takes a niche ’
                ‘approach to piercing coining the term fine body art creating ’
```

```
                'specialist of the moment jewellery of the highest quality her '
                'designs are ornately decorative featuring fine jewellery '
                'elements and decadent embellishments expanding her collection '
                'tash continues to win the hearts of body art enthusiasts who '
                'crave a refined aesthetic',
 'availability': None,
 'brand': 'maria tash',
 'manufacturer': None,
 'price': None,
 'priceCurrency': None,
 'title': None
}
{
 'name': '3d dropout adapter 10mm spacer',
 'description': 'thule 3d dropout adapter 10mm spacerenables you to attach a '
                'bicycle trailer kit to bicycles with hooded dropoutsfeatures '
                'simple and easy installation',
 'availability': None,
 'brand': None,
 'manufacturer': None,
 'price': None,
 'priceCurrency': None,
 'title': None
}
```

4.3.4 Music Brainz

The Music Brainz dataset is from the University of Leipzig [38]. The original dataset is synthetically generated using DaPO [35]. We processed this dataset by reformatting the data but otherwise leave it unchanged. Below we give two example records from the dataset.

```
{
 'title': 'Particle/Wave',
 'album': 'Evolver',
 'number': '2',
 'length': '381000',
 'artist': 'Lunchbox',
 'year': '2002',
 'language': 'English'
}
{
 'title': 'Ludwig van Beethoven - Sonata for Piano No. 10 in G major, Op. 14 '
          'No. 2: I. Allegrro',
 'album': 'The Piano - The ultimate piano collection of the century',
 'number': '1',
 'length': '428',
 'artist': None,
 'year': '08',
 'language': 'English'
}
```

4.3.5 North Carolina Voters

The North Carolina Voters dataset is from the University of Leipzig [38]. The original dataset is synthetically generated using GeCo [13]. We processed this dataset by reformatting the data but otherwise leave it unchanged. Below we give two example records from the dataset.

```
{
 ‘givenname’: ‘annie’,
 ‘surname’: ‘johmson’,
 ‘suburb’: ‘warremton’,
 ‘postcode’: ‘27589’,
}
{
 ‘givenname’: ‘dorthy’,
 ‘surname’: ‘yeager’,
 ‘suburb’: ‘jamestown’,
 ‘postcode’: ‘2728z’,
}
```

4.4 Downsampling

As discussed earlier, a key challenge in creating BigGoat is how to downsample Tables A and B to Tables A' and B' such that the recall estimated for a blocker Q using A' and B' is close to the recall of Q executed over A and B.

In this section we examine this challenge in detail and develop our downsampling algorithm. We begin by examining how to estimate recall for independent blockers. Next, we examine estimating recall for dependent blockers. We then present our downsampling procedure in detail. In a later section we present experimental results demonstrating the effectiveness of our solution at estimating recall for a variety of blocking methods and datasets.

4.4.1 Estimating Recall of Independent Blockers

We distinguish independent and dependent blockers. Intuitively, a blocker is independent if when given two tuples it can decide whether to return them (i.e., put them in the output of the blocker) *based solely on those two tuples*.

If this decision also depends on other tuples, then the blocker is dependent. For example, a top-k blocker such as Sparkly is dependent since whether a tuple pair makes it into the top k does depend not just on that tuple pair but also on other tuple pairs.

Definition 1. *Let $\mathcal{U}$ be all possible tuples. We define an independent blocker Q to be a function that takes as input two tuples and outputs true if the tuples are predicted to match and false otherwise. Formally, $Q : \mathcal{U} \times \mathcal{U} \rightarrow \{true, false\}$.*

Example 4.4.1. *Let $Q(a, b) =$ Jaccard3Gram$(a, b) \geq 0.7$ then it is an independent blocker according to Definition 1.*

We begin by considering how to estimate recall for an *independent blocker* using a sample. Let A and B be tables of tuples, $G \subseteq A \times B$ be the set of gold matches, and Q be an independent blocker. Let *Recall*(Q, S, G) be the recall of Q over a set of tuple pairs $S \subseteq A \times B$, formally, *Recall*$(Q, S, G) = |\{(a, b)|(a, b) \in S \wedge Q(a, b) =$ *true*$\}|/|S \cap G|$.

4.4.1.1 Random Sampling

We first examine baseline random sampling and analyze the sample size required to accurately estimate the recall of an independent blocker. Let $S \subseteq A \times B$ be a random sample and our estimated recall be *Recall*(Q, S, G). We can model *Recall*(Q, S, G) as an estimator of a Bernoulli random variable, where the random variable takes a value of 1 if $(a, b) \in G \wedge Q(a, b) =$ *true* and 0 otherwise.

Since S is an i.i.d. random sample from $A \times B$, the expected value of *Recall*(Q, S, G) is *Recall*$(Q, A \times B, G)$. Formally, $E\left[|\{(a, b)|(a, b) \in S \wedge Q(a, b) = \textit{true}\}|/|S \cap G|\right] = |\{(a, b)|(a, b) \in A \times B \wedge Q(a, b) = \textit{true}\}|/|G|$, where $E[\ldots]$ is the expected value. In other words, *Recall*(Q, S, G) is an unbiased estimator of *Recall*$(Q, A \times B, G)$.

Additionally, this means that *Recall*(Q, S, G) is normally distributed, which allows us to compute a confidence interval. Suppose we want to *Recall*(Q, S, G) to be within $.025$ of *Recall*$(Q, A \times B, G)$ 95% of the time. Let $\hat{t}$ be the estimated recall, z be the standard z-score, α be the confidence interval width, and n be the sample size. We can calculate the confidence interval of our recall estimate with the formula below,

$$\hat{t} \pm z_{1-\alpha/2} \cdot \sqrt{\frac{\hat{t}(1-\hat{t})}{n}}$$

For a 95% confidence interval we then get,

$$\hat{t} \pm z_{.975} \cdot \sqrt{\frac{\hat{t}(1-\hat{t})}{n}}$$
$$\hat{t} \pm 1.96 \cdot \sqrt{\frac{\hat{t}(1-\hat{t})}{n}}$$

We can then set this to our desired confidence interval width (0.025) and solve for the fraction of gold matches that we need in our sample. Let $t =$ *Recall*$(Q, A \times B, G)$, and x be the fraction of $A \times B$ that we use for our sample size. Note if we take $|A \times B| \cdot x$ tuples from $A \times B$, our effective sample size for estimating recall, n, is $|G| \cdot x$, since we only use true matches to estimate recall. Substituting from the formula above we get,

$$.025 = 1.96 \cdot \sqrt{\frac{\hat{t}(1-\hat{t})}{n}}$$
$$.025 = 1.96 \cdot \sqrt{\frac{t(1-t)}{|G| \cdot x}}$$

We can then rearrange to solve for the fraction of gold matches we need in the sample (x),

$$
\begin{aligned}
.025 &= 1.96 \cdot \sqrt{\frac{t(1-t)}{|G| \cdot x}} \\
\frac{.025}{1.96} &= \sqrt{\frac{t(1-t)}{|G| \cdot x}} \\
\left(\frac{.025}{1.96}\right)^2 &= \frac{t(1-t)}{|G| \cdot x} \\
\left(\frac{1.96}{.025}\right)^2 &= \frac{|G| \cdot x}{t(1-t)} \\
\left(\frac{1.96}{.025}\right)^2 \cdot t(1-t) &= |G| \cdot x \\
\left(\frac{1.96}{.025}\right)^2 \cdot \frac{t(1-t)}{|G|} &= x \\
\left(\frac{1.96}{.025}\right)^2 \cdot \frac{t(1-t)}{|G|} &= x
\end{aligned}
$$

Let $N = |A| = |B|$. If we assume that $|G|$ grows linearly with $|A|$ and $|B|$, we can then solve for the sample size,

$$
\begin{aligned}
|S| &= x \cdot |A \times B| \\
|S| &= O(\frac{1}{|G|}) \cdot |A \times B| \\
|S| &= O(\frac{1}{N}) \cdot O(N^2) \\
|S| &= O(\frac{N^2}{N}) \\
|S| &= O(N)
\end{aligned}
$$

Therefore, we get that $|S|$ grows linearly with $|A|$ and $|B|$, that is, as A and B become large, our sample size will also have to be large in order to maintain the same confidence interval. This means that randomly sampling will work for small A and B but become increasingly expensive

for larger datasets. For example, we $|A| = |B| = 100M$ and $|G| = 100M$, and our true recall is 0.95, we can solve for the sample size,

$$
\begin{aligned}
|S| &= x \cdot |A \times B| \\
|S| &= 6146.55 \cdot \frac{t(1-t)}{|G|} \cdot |A \times B| \\
|S| &= 6146.55 \cdot \frac{0.0475}{10^8} \cdot 10^{16} \\
|S| &= 6146.55 \cdot \frac{0.0475}{10^8} \cdot 10^{16} \\
|S| &= 2.91 \cdot 10^{-6} \cdot 10^{16} \\
|S| &= 2.91 \cdot 10^{10}
\end{aligned}
$$

That is, for a $|A| = |B| = 100M$ and $|G| = 100M$, we need a sample of over $10B$ records to be labeled in order to achieve the desired confidence interval.

4.4.1.2 Improved Random Sampling

While random sampling can work for estimating recall of an independent blocker Q for small A and B, a natural question is whether we can improve upon this solution. We observe that for estimating recall, only pairs in G affect our estimate. Therefore, if we can eliminate non-matching pairs from our sample we can reduce the size of our sample without reducing the accuracy of our recall estimate. To achieve this, we modify our random sampling by first running Sparkly Auto and then taking a random sample of the resulting candidate set.

Specifically, given a table A and B and blocker Q, we run Sparkly Auto over A and B with a fixed k to get a candidate set E and take a random sample of E to get S. We then estimate recall of Q over A and B as ***Recall***$(Q, S, S \cap G)$.

The intuition here is that if using Sparkly Auto we would mostly eliminate non-matches, then most of the matches are still in the candidate set E, which is a lot smaller than $A \times B$. So by taking

a random sample S from E, we can work with a much smaller sample S and yet still obtain an accurate estimation of recall for A and B.

The key challenge of this new random sampling procedure is how to choose the k value for Sparkly Auto to produce E. Choosing the k value is difficult, requiring domain knowledge and expertise, in practice we just set it to 10.

Continuing our analysis of the size of S in our improved random sampling, if we assume that $G \subseteq E$ then we can replace $A \times B$ with E,

$$
\begin{aligned}
|S| &= x \cdot |E| \\
|S| &= x \cdot O(|B| \cdot k) \\
|S| &= O(\frac{1}{N}) \cdot O(Nk) \\
|S| &= O(\frac{N}{N}) \\
|S| &= O(1)
\end{aligned}
$$

Therefore, our improved random sampling allows us to take a constant sample size, making it practical for large A and B.

4.4.2 Estimating Recall of Dependent Blockers

As discussed earlier, a dependent blocker cannot just look at a tuple pair and decide that it is likely to match and so would output that tuple pair. Instead, this decision also depends on other tuple pairs. A classic example of such blockers is a top-k blocker: we cannot decide if a tuple pair is in the top k without examining also many other tuple pairs.

Formally, we capture this by saying that a dependent blocker takes as input two tables, i.e., sets of tuples, and outputs a set of tuple pairs:

Definition 2. *Let $\mathcal{U}$ be all possible tuples. We define a dependent blocker F to be a function that takes in two sets of tuples and outputs a set of tuple pairs which are predicted to match, formally, $F : \mathcal{P}(\mathcal{U}) \times \mathcal{P}(\mathcal{U}) \rightarrow \mathcal{U} \times \mathcal{U}$.*

Example 4.4.2. *Let* $F(A, B) = \{(a, b)|b \in B \wedge a \in$ *BM25TopK*$(A, b, k)\}$*, where* *BM25TopK*$(A, a, b, 10) =$ *true if* a *is in the top-*k *records from* A *for* b*. This is a dependent blocker according to Definition 2.*

We now examine estimating recall for *dependent blockers* (Definition 2). First, we examine whether our random sampling approach described so far can work for dependent blockers. Next, we discuss the modifications needed to efficiently estimate the recall of dependent blockers. Finally, we examine how to downsample for a common category of dependent blockers: top-k blockers.

4.4.2.1 Random Downsampling for Dependent Blockers

First we examine whether our improved random sampling as described earlier can work for dependent blockers. Let A and B be tables of tuples, $G \subseteq A \times B$ be the gold matches, and F be a dependent blocker. Recall that our improved random sampling procedure works as follows:

1. Apply Sparkly Auto to Tables A and B (with a fixed k) to obtain a set of tuple pairs E.
2. Randomly sample from E to obtain a sample S.
3. Apply the blocker Q to S, then use $G \cup S$ to estimate the recall of Q over S.

If the blocker is independent, then we know how to execute Step 3, because we can apply the blocker Q to each tuple pair in S. However, if the blocker is dependent, then we cannot execute Step 3, because we cannot apply blocker Q to each tuple pair in S. Recall that in order to judge whether a tuple pair in S is likely to match, blocker Q needs access to the entire tables A and B (for example, to decide if a tuple pair is in the top k).

To address this problem, we can modify our recall computation to be *Recall*$(F, S, G) = |F(A, B) \cap S \cap G|/|S \cap G|$. That is, run F over the entire A and B and then only use pairs in S to estimate recall. While this is an unbiased estimator of the recall of F, it requires that we compute $F(A, B)$, the computation that we wanted to avoid by taking the downsample. So our random sampling procedure as is not well suited for dependent blockers.

4.4.2.2 Downsampling for Dependent Blockers

Thus, we must modify our sampling procedure such that we can estimate the recall of F without computing $F(A, B)$. Since our dependent blocker takes two tables as input, we modify our downsampling procedure to produce two tables $A' \subseteq A$ and $B' \subseteq B$.

Then our goal is: given tables A and B, gold G, and a dependent blocker F, produce $A' \subseteq A$ and $B' \subseteq B$, such that $|F(A', B') \cap G|/|A' \times B' \cap G| \approx |F(A, B) \cap G|/|G|$. The key challenge is developing a sampling procedure to produce Tables A' and B' such that it works for any dependent blocker F. In practice, this is not possible since dependent blockers vary greatly in how the output is computed. Instead of trying to develop a sampling procedure for *all* dependent blockers, we focus on top-k based blockers since they are one of the most common types of dependent blockers used in both academia and industry.

4.4.2.3 Estimating Recall for Top-k Blockers

Let A and B be the two tables to be blocked. We consider a top-k blocker $Q(A, B, k)$ such that for each tuple $b \in B$, Q finds the top-k tuples in A (that is, the k tuples in A with the highest similarity score to tuple b), then pairs b with these tuples and outputs the pairs.

Suppose we have downsampled A and B into smaller tables A' and B', respectively. Then we can estimate the recall of $Q(A', B', k)$ using the set of gold matches G. We want this estimated recall to be as close as possible to the actual recall of $Q(A, B, k)$.

Let us examine under what conditions we may achieve this. Suppose we have:

$$A = \{a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8\}$$
$$B = \{b_1, b_2, b_3, \dots\}$$
$$G = \{(a_4, b_1), (a_1, b_2), (a_6, b_3)\}$$

Now suppose the top-k blocker Q produces the following output:

$$Q(A, b_1, 5) = (a_2, a_3, a_8, a_4, a_7)$$
$$Q(A, b_2, 5) = (a_1, a_4, a_5, a_8, a_3)$$
$$Q(A, b_3, 5) = (a_7, a_3, a_6, a_2, a_8)$$

That is, for tuple b_1, Q returns the top-5 tuples from A to be the list $(a_2, a_3, a_8, a_4, a_7)$ (sorted in descending order of the similarity score), and so on. Then we can easily compute the recall of Q on A and B with respect to various k:

$$\text{Recall @ } 1 = 0.33$$
$$\text{Recall @ } 2 = 0.33$$
$$\text{Recall @ } 3 = 0.66$$
$$\text{Recall @ } 4 = 1.0$$
$$\text{Recall @ } 5 = 1.0$$

Now let us consider the smaller downsampled tables A' and B'. Suppose the downsampling is such that we have removed a_1 and a_4 (which have matches in B) from A'. Then we increase the variance of our estimated recall since we have fewer true matches to estimate recall with.

Specifically, we underestimate recall for $k = 1$ and $k = 2$ and overestimate recall for $k = 3$.

$$A' = \{a_2, a_3, a_5, a_6, a_7, a_8\}$$
$$B' = \{b_1, b_2, b_3, \dots\}$$
$$G' = \{(a_6, b_3)\}$$
$$Q(A', b_1, 5) = (a_2, a_3, a_8, a_7, a_1)$$
$$Q(A', b_2, 5) = (a_5, a_8, a_3, a_6, a_2)$$
$$Q(A', b_3, 5) = (a_7, a_3, a_6, a_2, a_8)$$

Estimated Recall @ $1 = 0.0$	Actual Recall @ $1 = 0.33$
Estimated Recall @ $2 = 0.0$	Actual Recall @ $2 = 0.33$
Estimated Recall @ $3 = 1.0$	Actual Recall @ $3 = 0.66$
Estimated Recall @ $4 = 1.0$	Actual Recall @ $4 = 1.00$
Estimated Recall @ $5 = 1.0$	Actual Recall @ $5 = 1.00$

If we remove a_2 and a_3 (which do not have matches in B) from A', we then overestimate the recall since some of the true matches are being produced at a lower k value. Specifically, we overestimate recall for $k = 3$.

$$A' = \{a_1, a_4, a_5, a_6, a_7, a_8\}$$
$$B' = \{b_1, b_2, b_3, \dots\}$$
$$G' = \{(a_4, b_1), (a_1, b_2), (a_6, b_3)\}$$
$$Q(A, b_1, 5) = (a_3, a_8, a_4, a_7, a_1)$$
$$Q(A, b_2, 5) = (a_1, a_4, a_5, a_8, a_7)$$
$$Q(A, b_3, 5) = (a_7, a_1, a_6, a_8, a_5)$$

Estimated Recall @ $1 = 0.33$	Actual Recall @ $1 = 0.33$
Estimated Recall @ $2 = 0.33$	Actual Recall @ $2 = 0.33$
Estimated Recall @ $3 = 1.0$	Actual Recall @ $3 = 0.66$
Estimated Recall @ $4 = 1.0$	Actual Recall @ $4 = 1.00$
Estimated Recall @ $5 = 1.0$	Actual Recall @ $5 = 1.00$

The above examples suggest two properties that we should try to preserve when we downsample the original tables. First, as with independent blockers, we must preserve enough matching pairs in $A' \times B'$ to keep the variance of our estimate low. Second, unlike independent blockers, we must also preserve similar non-matching tuple pairs to prevent overestimating recall, where a similar non-matching pair is a pair of tuples that do not match but have a high similarity score. For example, "iPhone 14 Pro red" and "iPhone 15 Pro red" is a similar non-matching pair since the product titles share many words but refer to different products.

4.4.2.4 Clustering for Downsampling

Now that we understand the two key properties of the data we must preserve to accurately estimate the recall of top-k blockers, we can begin to develop a sampling procedure. Intuitively, a solution to the above problems is to cluster all tuple in $A \cup B$. This will give us clusters of matching pairs and pairs that are similar but non-matching. We can then sample from these clusters.

The key challenge of using clustering for downsampling is scalability, as the majority of clustering algorithms are designed to run on a single machine and/or have quadratic (or greater) runtime complexity. This means that most clustering algorithms are not feasible for large datasets. As a result, we consider three clustering methods: K-Means, Power Iteration Clustering, and Hierarchical Agglomerative Clustering. For each method we give a brief description followed by an analysis of its scalability.

K-Means: K-Means clusters points in Euclidean space by iteratively refining k cluster centroids. Since our records do not exist in Euclidean space, a sensible baseline would be to use a text embedding model to convert each tuple in to a d-dimensional vector which can then be used in the

K-means algorithm. For current state-of-the-art embedding models typically $d \geq 300$. If $d = 300$, converting 100M records to embeddings produces 111 GB of data, requiring significant hardware to run. Additionally, the runtime complexity of K-Means can also pose a significant challenge. While the runtime complexity of K-Means is linear in the number of input tuples, specifically $O(dNk)$, where N is the number of input tuples and k is the number of clusters, this assumes that k is constant. In practical applications we likely want k to scale with the number of input tuples to keep the output clusters of manageable size, if $k = O(N)$ then the runtime complexity becomes $O(dN^2)$. The large memory footprint combined with quadratic runtime makes K-means impractical for our use case.

Power Iteration Clustering: This method [44] takes a sparse graph as input and learns a low-dimensional embedding for each node (that is, record) in the graph and then uses these learned low dimensional embeddings to cluster the nodes. We tested a Spark implementation of the algorithm but found that it produced low quality clusters. So we drop it from consideration.

Hierarchical Agglomerative Clustering (HAC): This method is a greedy algorithm that takes as input a set of objects $U \subseteq \mathcal{U}$, a similarity function $f : \mathcal{U} \times \mathcal{U} \to \mathbb{R}$ and greedily merges clusters together until a stopping condition is met. While the naive HAC algorithm is $O(N^3)$, we modify it to create a novel scalable clustering algorithm which we use for our downsampling procedure.

4.4.2.5 Modified HAC for Downsampling

There are two key components of HAC which affect the runtime complexity: the linkage (cluster merging) criteria and the stopping criteria. Below we describe how we modify these components to create a scalable clustering algorithm that we use for our downsampling procedure.

Linkage Criteria: There are a variety of linkage criteria for HAC; the most common are average, complete, WARD, and single, each of these linkage criteria define how to compute the similarity score between two clusters, which is used to determine the order clusters are merged. We use single linkage since, unlike other common kinds of linkages, it does not require recomputing the similarity between clusters after each iteration, hence allowing us to save significant time. While single

linkage saves time, it still requires an $O(n^2)$ computation for computing the similarity scores between all pairs. To reduce the runtime complexity, we leverage Sparkly Auto to efficiently compute only the top-k highest scoring records from A for each record in B (where k is a hyperparameter, currently set to 10) and set the score for all other pairs to be zero.

Stopping Criteria: For stopping criteria we set a limit on the cluster sizes. Specifically, given two cluster c_1, c_2 if $|c_1 \cup c_2|$ is larger than a predetermined maximum size t (current set to 32), we skip merging the clusters. This design decision was made for two reasons. First, from a scalability standpoint this only requires a constant time check, which is significantly faster than other common stopping criteria such as cluster purity metrics.

Second, when using single linkage, even when limited to considering only top-k most similar objects, it is very easy to end up with a few very large clusters. Not only does this lead to poor quality clusters, it also makes it impossible to achieve the desired sample size without splitting the clusters (as we discuss below). By having a limit t on the cluster size, we can ensure that our actual sample size is within t records of the desired sample size (as we discuss below).

4.4.3 Our Downsampling Procedure

We now describe our downsampling procedure (Algorithm 4.2) in detail. Our procedure takes as input two tables A, B, sample size $N < |A| + |B|$, $k \in \mathbb{Z}^+$, and cluster size limit $t \in \mathbb{Z}^+$ and outputs two tables $A' \subseteq A$ and $B' \subseteq B$ where $|A'| + |B'| = N \pm t$. Let W be a set of clusters, where $v \in A \cup B, W[v]$ is the cluster that record v currently belongs to.

- Initialize the clusters, W, where each record in A and each record in B belong to its own cluster
- Run Sparkly Auto over A and B to get top-k candidates for each record $b \in B$ and union these top-k candidates into a candidate set C
- Sort C into descending order by score to get C'
- For each pair (a, b) in C'

  - Merge $W[a]$ and $W[b]$ into a single cluster if $|W[a]| + |W[b]| \leq t$.

- Initialize sample S as an empty set
- For each pair (a, b) in C'
  - If $W[a] = W[b]$ and $|S| < N$, add $W[a]$ to the sample S
- Create A' by collecting all the records from A that are in the sample S, that is, $A' = A \cap S$.
- Create B' by collecting all the records from B that are in the sample S, that is, $B' = B \cap S$.
- Return tables A' and B'

We note that we add the clusters to the sample in sorted for C' since the higher scoring pairs are more likely to be matches. As explained above, in order to estimate recall, we must preserve enough matching pairs to keep the variance of our recall estimate low, therefore taking the clusters in sorted order ensure that there is at least one likely matching pair in the cluster. Additionally, we only add clusters where $W[a] = W[b]$ since adding clusters when $W[a] \neq W[b]$ effectively doubles t and therefore would increase the range of the output size from $N \pm t$ to $N \pm 2t$.

4.4.4 Downsampling Procedure Example

Below we give a small example of the clustering algorithm used in our downsampling procedure. For this example we have set the max cluster size t to be 4, and there are a total of 9 records, with ids $0, \ldots, 8$. Table 4.2 shows C' sorted in descending order of score.

Below we show the state of the clusters after each step in the algorithm.

id1	id2	score
2	8	9.59
3	0	8.67
3	2	8.50
3	1	8.20
1	5	7.18
4	3	6.70
0	1	5.98
4	5	5.09
2	6	3.38
0	2	3.19
1	4	0.17

Table 4.2: C' sorted in descending order of score.

Initial state, all tuples belong to their own cluster. Clusters $= \{0\}, \{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}, \{8\}$
Merge 2 and 8, first row in C' sorted in descending order of score. Clusters $= \{0\}, \{1\}, \{2, 8\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}$
Merge 3 and 0. Clusters $= \{0, 3\}, \{1\}, \{2, 8\}, \{4\}, \{5\}, \{6\}, \{7\}$
Merge 3 and 2. Clusters $= \{0, 2, 3, 8\}, \{1\}, \{4\}, \{5\}, \{6\}, \{7\}$
Merge 3 and 1, skipped because resulting cluster would exceed max cluster size, $t = 4$. Clusters $= \{0, 2, 3, 8\}, \{1\}, \{4\}, \{5\}, \{6\}, \{7\}$
Merge 1 and 5. Clusters $= \{0, 2, 3, 8\}, \{1, 5\}, \{4\}, \{6\}, \{7\}$
Merge 4 and 3, skipped because resulting cluster would exceed max cluster size, $t = 4$. Clusters $= \{0, 2, 3, 8\}, \{1, 5\}, \{4\}, \{6\}, \{7\}$
Merge 0 and 1, skipped because resulting cluster would exceed max cluster size, $t = 4$. Clusters $= \{0, 2, 3, 8\}, \{1, 5\}, \{4\}, \{6\}, \{7\}$
Merge 4 and 5. Clusters $= \{0, 2, 3, 8\}, \{1, 4, 5\}, \{6\}, \{7\}$
Merge 2 and 6, skipped because resulting cluster would exceed max cluster size, $t = 4$. Clusters $= \{0, 2, 3, 8\}, \{1, 4, 5\}, \{6\}, \{7\}$
Merge 0 and 2, skipped because tuples are already in same cluster. Clusters $= \{0, 2, 3, 8\}, \{1, 4, 5\}, \{6\}, \{7\}$
Merge 1 and 4, skipped because tuples are already in same cluster. Clusters $= \{0, 2, 3, 8\}, \{1, 4, 5\}, \{6\}, \{7\}$
C' exhausted, terminate. Clusters $= \{0, 2, 3, 8\}, \{1, 4, 5\}, \{6\}, \{7\}$

4.4.5 Evaluation

We now evaluate our downsampling procedure. For each blocking method and dataset we compare the recall on the full dataset to the recall estimated using the output of the downsampling procedure. Our goal is to see if our downsampling method will produce Tables A', B' such that $\textit{Recall}(Q, A \times B, G) \approx \textit{Recall}(Q, A' \times B', G')$.

Datasets: We used the five datasets from the BigGoat benchmark (see Table 4.1). For each dataset we manually select a set of input columns that are used across all blocking methods and samples. Specifically, for Big Citation we use the "title" column, for FEC we use the "name" column, for WDC we use the "name" column, for Music Brainz we use the "title" and "artist" columns, and for North Carolina Voters we use the "givenname" and "surname" columns. The sample size for each dataset was set to either 500,000 records or 10% of the dataset, whichever is smaller. For non-deterministic sampling procedures we produced 10 samples and plotted the median recall at each point.

Sampling Methods: We compare two sampling methods in our evaluation, Falcon and our sampling method as described in Section 4.4.3. Falcon is an adapted version of the sampling procedure described in [19] and is the most advanced sampling method for this setting that we are aware of. Specifically, we modified the sampling procedure by computing the likely matching pairs using Sparkly Manual, setting the y parameter to 10, and converting the output pairs into two tables A' and B'.

Blockers: We use three common blockers for our evaluation: Sparkly Manual, Vector Top-k, and Jaccard. For Sparkly Manual, we use the algorithm described in Section 2.2.3 without modification. For Vector Top-k, each record was converted into an embedding by concatenating the selected input columns together and converting the resulting string into an embedding using BGE-small (384 dimensional embedding). The top-k computation was brute-force to avoid any effects of approximate indexing techniques. For Jaccard, we used a word-based tokenizer. Note we only ran Sparkly Manually for FEC, since running Vector Top-k was estimated to take over a week, and Jaccard produced an extremely large candidate set which caused out-of-memory errors.

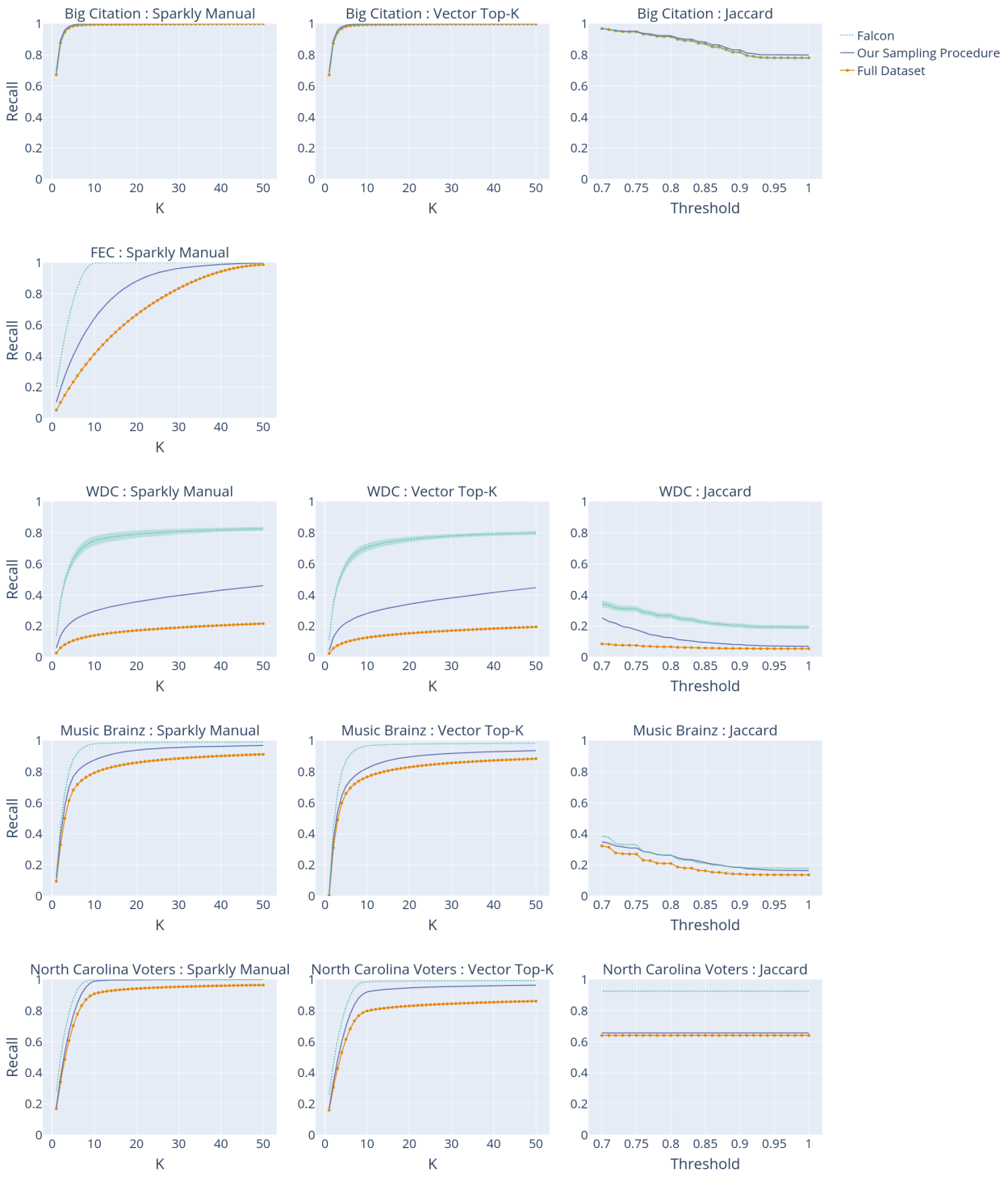

Figure 4.1: Comparision of recall estimates for downsampling methods.

Discussion: We now discuss our experimental results. We begin with comparing our downsampling method to Falcon, and then discuss the relatively low performance observed on WDC. Despite outperforming Falcon, our sampling method still has a significant difference between the estimated recall and actual recall on WDC.

In Figure 4.1 we compare the recall estimates using downsampling methods to the actual recall on the full dataset. The x-axis is the k value for Sparkly Manual and Vector Top-K or the threshold for Jaccard. The y-axis is the recall. The title of each subplot is the dataset that was downsampled and the blocker that was run.

For example, the top-left subplot contains recall numbers for Sparkly Manual run on the Big Citation dataset. The orange line is the recall of the blocking method on the full dataset. The solid blue line is the recall estimated using our downsampling procedure. The green dotted line is the recall estimated using the Falcon downsampling procedure.

From Figure 4.1, we find that our sampling procedure either matches or outperforms the Falcon downsampling procedure. That is, across all datasets and blockers, the recall estimate produced using our sampling method is closer to the recall on the full dataset than the Falcon recall estimate. For Big Citation and Music Brainz, Jaccard, the recall estimates are very similar, within approximately 1% of the recall on the full dataset. For all other datasets, our sampling method produces a significantly more accurate recall estimate for all blocking methods tested, in some cases reducing the estimate error by over 50%.

While our sampling method can be used to produce more accurate recall estimates when compared to Falcon, on some datasets our estimate is still significantly different from the actual recall. For example, on WDC for both Sparkly Manual and Vector Top-k, our estimated recall is roughly double that of the actual recall. We attribute this difference to WDC being a relatively hard dataset to match as shown by the low recall on the full dataset of the blockers tested.

4.5 Upsampling

In the BigGoat benchmark we are focused on testing the scalability of blockers using data as realistic as possible. However, realistic data is limited in size and thus limits the scale we

can test. To address this limitation, we have developed a data generation procedure that takes an existing dataset and "upsample", that is, generates an arbitrarily large dataset that we can then use to evaluate the scalability of blockers.

A baseline solution to generate datasets of arbitrary size is to simply duplicate the input tables. For example, to generate a dataset $10\times$ the size of an input dataset we could just duplicates the tables $10\times$ and update the gold accordingly. This approach is unsatisfactory since it will skew the runtime and output size of many blockers. For example, Jaccard with a threshold will produce a candidate set $100\times$ the size of the original candidate set and we are likely to see a similar increase in runtime.

The key challenge of generating datasets is to generate data which will have a similar runtime to a real data source. In principle there is no way to guarantee that a generated dataset will reflect a real runtime for any given blocker, since each blocker's runtime will be affected by different properties of the dataset. As a result, we focus on two key aspects of datasets that can affect the runtime of many blockers: the average number of matches per record in the dataset, and the distribution of values in the dataset.

The rest of this section is structured as follows. First we give procedures for generating records with matches and records without matches. Next, we present our complete data generation procedure to upsample a dataset. Finally, we present our evaluation of the data generation procedure and discuss the results.

4.5.1 Generating Records with Matches

We consider generating records with matches while preserving the average number of matches per record in the generated dataset. In order to do this, we create a procedure to generate matching records. This problem is inherently difficult since what it means for two records to match is context dependent and frequently not well-defined. For example, we may or may not consider "iPhone 15 Pro red" and "iPhone 15 Pro silver" to be a match, depending on the context. The baseline solution would be to simply duplicate certain records, however this is not realistic. Instead, we generate

new matching records by taking existing records with matches and perturbing them in ways that mimic common data quality issues.

When perturbing records we utilize three different functions which mimic common forms of data quality issues: `concat`, `typo`, and `truncate`. Function `concat` takes a record and returns a new record with a randomly selected field concatenated to the 'main field' of the record. Here the 'main field' is the field which contains the key information about the entity the record refers to, e.g., person name, paper title, or product title.

Example 4.5.1.

$$x = \{\texttt{'name' : 'John Smith', 'zipcode' :12345}\}$$
$$\texttt{concat}(x) = \{\texttt{'name' : 'John Smith 12345', 'zipcode' : null}\}$$

Function `typo` takes a record and return a new record with one to three characters swapped in a randomly selected field.

Example 4.5.2.

$$x = \{\texttt{'name' : 'John Smith', 'zipcode' :12345}\}$$
$$\texttt{typo}(x) = \{\texttt{'name' : 'Jahn Smoth', 'zipcode' : 12345}\}$$

Finally, function `truncate` takes a record and returns a new record with up to 75% of a randomly selected field truncated.

Example 4.5.3.

$$x = \{\texttt{'name' : 'John Smith', 'zipcode' :12345}\}$$
$$\texttt{truncate}(x) = \{\texttt{'name' : 'John Smi', 'zipcode' : 12345}\}$$

To perturb a single record we apply two or three of the above functions at random to the record, to create a new record which matches the original but is not identical.

To generate records with matches in the new dataset we apply the following procedure (Algorithm 4.4). Our procedure takes as input two tables A and B, gold G, and a target number of matches N and outputs two upsampled tables A' and B', and gold G', where $|G'| \approx N$.

- Duplicate A, B, and G, to get A', B', and G'
- While $|G'| < N$
  - Select a random record, x, from $A \cup B$, that has at least one match
  - Generate a new record x' by randomly apply 2 or 3 functions from above
  - Update the gold G' with the new matches for x'
- Return A', B', and G'

4.5.2 Generating Records without Matches

In the second step of data generation, we add records with no matches to the new dataset to achieve the desired size. In order to maintain the distribution of values within each column while generating records with no matches, we develop a procedure to generate realistic records that do not match any records currently in the dataset. We decompose this problem into generating values for a given column, specifically numeric columns and string columns.

For numeric columns we generate values using the follow procedure. Given a numeric column A_i and number of values to generate N, we first fit a kernel density estimation (KDE) model M to the non-null values of A_i and compute q, the fraction of null values in A_i. Next, we generate N values, A_i', by randomly sampling from the distribution of M. Finally, we randomly set $\lfloor N \cdot q \rfloor$ values in A_i' to be null and return A_i'.

For string columns, we aim to preserve the distribution of tokens in the dataset, while adding some realistic variation to the data. To do this, we split the problem of generating a new string values into two steps: determining the number of tokens in the string and generating tokens.

Given a string column A_i and number of values to generate N, we first fit a KDE model M to the number of tokens in non-null strings of A_i and compute q, the fraction of null values in A_i. Next, we use an LLM with a few-shot prompt to generate a set of values S_i, where $|S_i| = \min(|A_i|, 50000)$.

We tokenize A_i and S_i and combine them to create a set of tokens P. We then generate N string values, A_i, by generating string lengths L_i using M and then for each l in L_i, we take a

random sample of the tokens in P and concatenate to produce a new string. Finally, we randomly set $\lfloor N \cdot q \rfloor$ values in A'_i to be null and return A'_i.

Finally, we use the following procedure to generate the records with no matches for a given table. Given a table T with m columns and number of records to generate N, for each column of T, $T_i, i \in 1, \ldots, m$, we generate a new column, T'_i, with N values using the data generation procedure above for the datatype of T_i. We then combine the generated columns $T'_1, \ldots T'_m$ to get create a new table T' and return T'.

4.5.3 Generating a New Dataset

We now describe our dataset generation procedure (see Algorithm 4.3). Given two tables A and B, a gold set G, and a scale factor $p > 1$, we first generate records with matches using the procedure in Section 4.5.1 with A, B, and G and a target number of matches of $|G| \cdot p$, to get A', B', and G'. Next, we generate records without matches for table A using the procedure in Section 4.5.2 with input A and a target size of $(|A| * p) - |A'|$ and concatenate the result to A' to get A''. We then repeat for B with input B and a target size of $(|B| * p) - |B'|$ and concatenate the result to B' to get B''. Finally, we return the generated dataset A'', B'', and G'.

4.5.4 Evaluation

We evaluate our upsampling procedure by comparing the runtime of various blockers on generated and real data. All datasets were run locally on a single machine with 16c/32t and 64 GB RAM running Ubuntu 22.04. All blockers are implemented using Spark.

Datasets: For evaluation we use four datasets from the Big Goat benchmark: FEC, WDC, Music Brainz, and North Carolina Voters. Note that we exclude Big Citation for this evaluation because it has fewer than 10M records and hence does allow a baseline for larger sample sizes.

Generation Procedure: For each dataset we take a random sample of 1M records and use that as the input data for our data generation procedure, to generate a new dataset which we in turn compare with the real data. For each dataset we manually selected the 'main field' to be the

'name' or 'title' column of the dataset. Recall that the 'main field' of the record (from Section 4.5.1), is the field which contains the key information about the entity the record refers to, e.g., person name, paper title, or product title.

Blockers: For our experiments we selected three blockers commonly used in entity matching: Sparkly, Jaccard, and Hash. For Sparkly, we used a word-based tokenizer and set k to 10. For Jaccard, we used a word-based tokenizer and set the threshold to 0.8. For Hash, we used a case-insensitive comparison, i.e., a case-insensitive equality join. All blockers used the same input columns which were manually selected for each dataset (see below).

Column Selection: For each dataset we manually selected the columns which the blockers used. Specifically, we selected columns that a user would likely pick to block the datasets with. For FEC, we selected the 'name' column, which contains the full name of the person. For WDC, we selected the 'name' column which contains the product name. For Music Brainz, we selected 'title' and 'artist', which contain the song title and artist name, respectively. We selected both columns since either column alone is unlikely to be sufficient to match the records in the dataset. Finally, for North Carolina Voters, we selected 'givenname' and 'surname' which contain the first and last name of the person, respectively. Again, we selected both columns since either column alone would be insufficient to match the records in the dataset.

Discussion: Figure 4.2 shows the runtime of various blockers on generated data (blue lines) and real data (orange lines) while varying the input data size. Each subplot corresponds to a single dataset and a blocker (specified in the title of the subplot). For example, the subplot in the top-left corner plots the runtime of Sparkly on the FEC dataset, which shows that Sparkly ran in 25 minutes on a 10M sample of the real data, and in 20 minutes on 10M sample of generated data.

From Figure 4.2 we see that overall the results are very inconsistent. In some cases, the runtimes on generated and real data are comparable (e.g., Sparkly on FEC), while in other cases they are significantly different (e.g., Jaccard on Music Brainz). In what follows we discuss the key differences between the real and generated datasets and use these differences to explain the runtime behavior we observed.

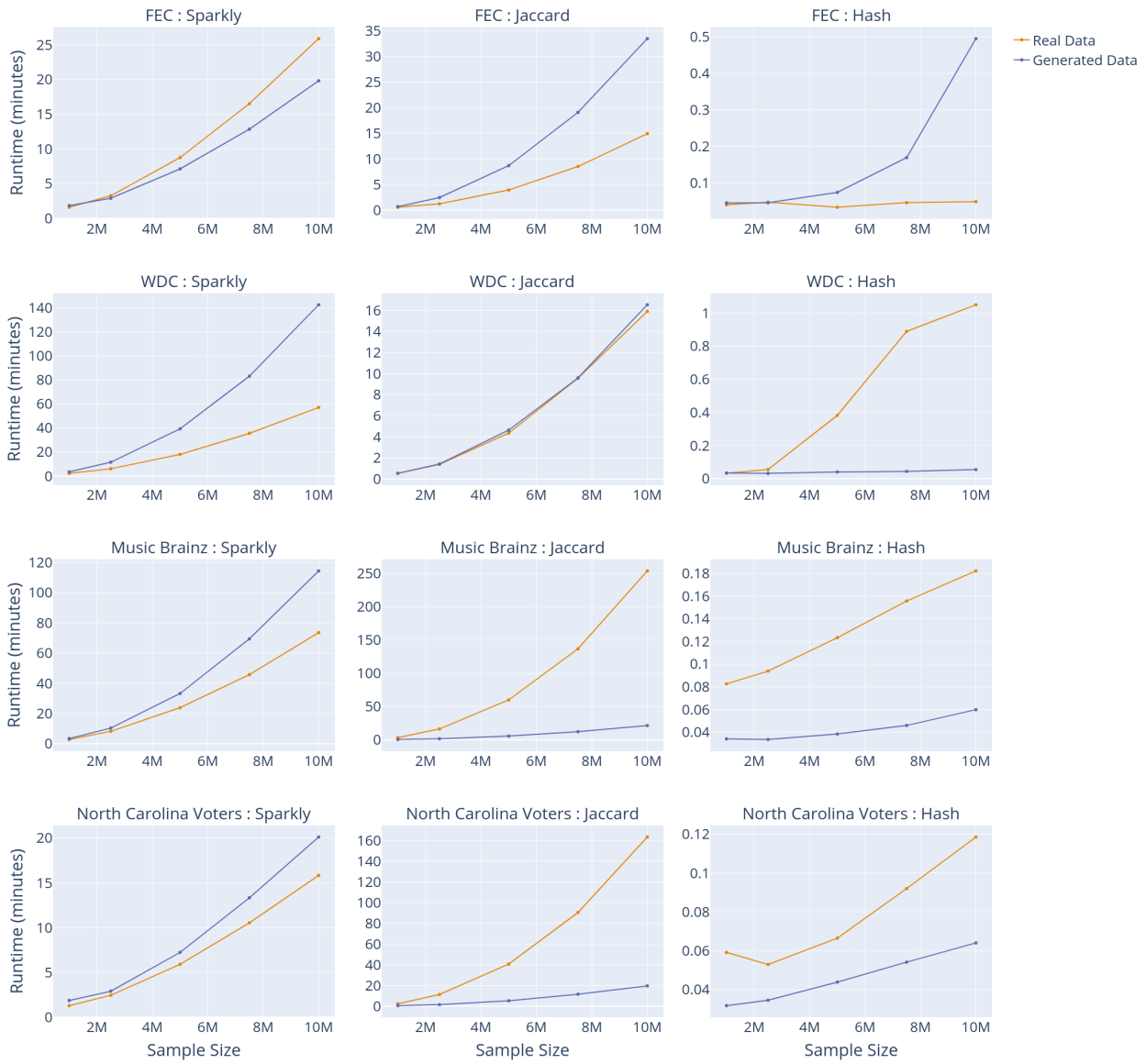

Figure 4.2: Runtime of various blockers on real and generated data

Differences in Real vs. Generated Data: To understand why the runtimes of real and generated data differed so much we compare the characteristics of the real and generated data. From this investigation we identified three major differences between the real and generated data.

Number of Unique Tokens: The first difference that we found in the data is the number of unique tokens. In particular, the real data has far more unique tokens than the generated data for a given

size. While we try to mitigate this issue by using an LLM to generate new values, in order to match the real data we would need to greatly increase the number of prompts that we run. Since running the LLM is by far the most computationally expensive part of the data generation process, this would significantly increase the overall runtime of the algorithm.

Number of Tokens per Field: The second difference we found was differences in the number of tokens per field. Specifically, the minimum number of tokens in the 'name' field of FEC is two in the real data (since it contains a full name), however a significant number of generated names have only a single token. This discrepancy is due to the use of the KDE model to choose the number of tokens in the string that we generate, since the KDE model can produce values outside the range of the input data.

Token Correlations and Patterns: The third difference between the real and generated data is the token correlations and patterns. Specifically, there are no correlations or patterns of tokens in the generated data since we are randomly sampling and concatenating tokens together to create string values while the real data does contain correlations and patterns. While it is difficult to quantify these patterns, we can observe the effects of them in the output size of the hash blocker. While the real data has far more unique tokens than the generated data, the output size of the hash blocker is larger on the real data.

Explanation: We now use the three key difference between the real and generated data (discussed above) to explain the runtime behavior observed in Figure 4.2.

Sparkly: For all datasets except for FEC, the generated data runs slower with Sparkly. As we discussed above, the tokens in the generated data are uncorrelated. Sparkly uses the Block Max WAND algorithm to evaluate top-k queries, which skips scoring records by computing upper bounds of the scores of the records.

This means that the runtime of Sparkly is sensitive to the scores since the probability of skipping a record increases with the difference of scores between records. Since the tokens in the generated data are uncorrelated, the difference between the top-k scores and the rest of the scores is less than that in the real data, making the skipping of the Block Max WAND algorithm less

effective. The exception to this is FEC, where we produce many single token strings which run very fast compared to the multi-token stings in the real data.

Jaccard: For Music Brainz and North Carolina Voters, we found that Jaccard runs significantly slower on the real data. We attribute this to far more records in the real data passing through the index filter and being scored since the tokens in the real data are correlated and hence more likely to pass filtering. For FEC, we found that the generated data is much slower, this is again due to the generated data containing many single token strings which cause far more records to pass filtering and be scored.

Hash: For the hash blocker we found that the real data ran slower on all datasets except for FEC. This is attributable to the number of records that are output by the blocker, the fewer records output the faster the run time. The difference in output size can be explained by the random generation or strings in the generated data. That is, it is very unlikely to produce the same string twice when randomly sampling even a few tokens. Again FEC is the exception since we produce many single token strings which have a fairly high likelihood of being produced twice during generation.

4.6 Related Work

Existing Entity Matching Benchmarks: There are numerous entity matching benchmark datasets that have been published, covering a variety of domains. The majority of datasets are focused on evaluating the precision and recall of entity matching systems and are generally 1M tuples or less. Some more recent work has incorporated different angles such as evaluating data integration pipelines [18], leveraging noisy labels [59], or combining evaluation on structured, semi-structured, and unstructured data [63].

The majority of benchmarks for entity matching are simply two tables A and B with a gold set $G \subseteq A \times B$ [38, 5, 20]. In these benchmarks G is assumed to be ground truth, that is, G is a complete set of matches between A and B with no false positives. These datasets are generally used to evaluate the precision and recall of an entity matching system and cover a variety of domains,

such as people, paper citations, products, and songs. Most of these datasets are 1M tuples or less, making them not suitable for evaluating the scalability of blockers.

The Alaska Benchmark [18] was created to address evaluation of complete data integration systems. It consists of a single table of 70K product descriptions extracted from 71 different data sources over three product types, cameras, monitors, and notebooks. The dataset was manually curated and has ground truth for schema matching and entity resolution, each of these tasks are further divided into easy, medium, and hard sets. For each of these tasks the user of the benchmark reports precision and recall for comparison with other solutions.

The WDC dataset is a product matching dataset from University of Mannheim [59] and is the largest entity matching dataset that we are aware of. In consists of 26M records that were extracted from the Common Crawl dataset. The labels for WDC were created using a combination of automatic labeling and manual labeling. The vast majority of labels were generated automatically via comparing extracted product identifiers (i.e. if two records shared the same product SKU they are labeled as a match). These automatically labeled pairs are used as the train set for entity matching algorithms. Additionally, a few thousand record pairs were manually labeled and are used as the test set for evaluation.

The Machamp benchmark is a recent work that combines the evaluation on structured, semi-structured, and unstructured data [63]. The Machamp dataset was created by combining datasets from [20] and [59] to create 7 datasets, each of which is a pair of tables. Each table is either structured (relational), semi-structured (JSON), or unstructured (text) data. The all datasets have ground truth provided and range in size from 1K to 70K. For each dataset users report precision and recall for comparison with other solutions.

Existing Downsampling Algorithms: Downsampling is a crucial part of creating the BigGoat benchmark. Specifically, we require a downsampling method that takes as input two tables A and B and outputs two tables A' and B' such that the recall of a given blocker Q executed over A and B is as close as possible the recall of Q executed over A' and B'. Many different methods of sampling have been developed, however we are not aware of any algorithms specifically designed for this task. The most relevant piece of work comes from [19], which gives an algorithm to create

a set of sample pairs to train a machine learning model. In particular, the algorithm takes as input two tables A and B, sample size N, and hyperparameter y, and outputs a sample $S \subseteq A \times B$. Briefly, the Falcon sampling method works by randomly sampling records from B to get B' where $|B'| = \lfloor N/y \rfloor$. For each record in $b \in B'$, the method creates a set of tuples S_b by selecting $y/2$ records from A that are likely to match b and $y/2$ randomly selected records from A. Finally, it constructs a set of sample pairs $S = \{(b, a)|b \in B' \ \wedge \ a \in S_b\}$ and returns S.

Existing Data Generation: There are many existing data generation solutions for benchmarking systems. These tools fall into two categories, tools for benchmarking query execution of databases and tools specifically created for benchmarking entity matching systems.

The two most widely used options for benchmarking scalability and performance of database engines are the TPC benchmarks and YCSB [17]. These tools have two main components, a data generation tool and a set of SQL queries to be executed over the data output by the data generation tool. To run the benchmark, the user calls the data generation tool, runs the SQL queries over the generated data, and then reports the size of the data generated and the runtime of the SQL queries. While these benchmarks are useful for testing database system, the data generated by these tools are not useful for entity matching. Specifically, the values in the tables output are random strings which don't contain semantically meaningful information in them. Since the runtime of most blocking solutions for entity matching systems are sensitive to the input values, these tools don't produce data that is useful for estimating runtime.

Over the past 30 years there have been a wide variety of data generation tools for entity matching that have been created [35, 51, 13, 36, 2, 34, 10]. Each of these solutions takes a small dataset as seed data to generate a larger dataset. We found that there were multiple shortcomings for these solutions. First, many required extensive configuration by the user. Second, many were specialized to specific domains (e.g. people, medical records), limiting their usefulness. Unfortunately, we were unable to locate source code for the majority of data generators, we were able to find source code for Febrl [10] and GeCo [13]. Febrl relies on lookup tables to generate data and hence is not suitable for comparison. GeCo we were unable to run successfully run the source since the

last time it was published was 2013. Additionally, none of these methods are tailored to estimate the runtime of blockers on larger datasets.

4.7 Conclusions

Benchmark: In this chapter we have created a novel scaling benchmark from realistic data sources that mirrors how blockers are created in the real world. We have identified three major directions for future work. First, our benchmark contains datasets up to 60M records. While this is far larger than any other benchmark dataset that we are aware of, there are industrial systems that must handle even larger data sets. Future work can improve the benchmark by adding even larger datasets.

Second, our benchmark covers a reasonable range of domains. However, there are still many domains that are not represented. Our experiments with downsampling (Figure 4.1) demonstrate how the domain of the data can significantly affect blocker recall and downsampling performance. Future work can expand the benchmark by adding new datasets that cover more domains and improve the quality of the blocker evaluation.

Finally, our blocking benchmark focuses on batch processing, that is, running blocking over an entire table or two tables. Another common application in industry is real time search (e.g., looking up an account for a single customer). Future work could expand the scope of the benchmark to include metrics for real time search by reporting query latency for a single search.

Downsampling: In this work we have created a novel downsampling procedure, specifically designed for estimating the recall of blockers. Future work can further refine downsampling procedures to produce more accurate recall estimates. Specifically, we see three possible directions for future work. First, our downsampling procedure can be refined in multiple ways such as: how records are scored, the clustering algorithm, and how clusters are selected.

Second, in our work we set the hyperparameters of our blocking solution manually. Specifically, we set k to 10 and maximum cluster size to 32. These parameters are likely suboptimal for some datasets and hence could benefit from a tuning procedure. Future work should consider

approaches for hyperparameter tuning to improve the performance of the sampling method across a wider variety of datasets.

Finally, our downsampling procedure was created specifically with top-k blockers in mind. Future work could develop procedures tuned for a specific blocking method. Such work would be able to account for the idiosyncrasies of particular blocking methods and likely be able to produce more accurate recall estimates.

Upsampling: In this work we have created a novel data generation algorithm to create arbitrarily large datasets for evaluating blockers. Our algorithm attempts to preserve two key characteristics of the input data that can affect the runtime of blockers: the average number of matches per record and the distribution of values in the dataset.

In our evaluation we found that the runtime of our generated data was inconsistent. In some cases it is close to real data, yet in many cases it was completely different. We have identified some key aspects of our generated data which are causing the observed differences. These aspects include the number of unique tokens in the data, the number of tokens per record, and the correlations between the tokens. While these are important for the blockers that we tested, there are almost certainly more aspects of the data that will affect the runtime of other blockers. Future work should identify other aspect of the data which affects the runtime of blockers and expose them as parameters, so that users can tune to adjust how difficult the generated data should be for their particular blocker or use case.

Algorithm 4.2 Pseudocode for our downsampling procedure

Input : Table A and B to be downsampled, sample size $N \in \mathbb{Z}^+$, initial top-k value $k \in \mathbb{Z}^+$, and cluster size limit $t \in \mathbb{Z}^+$

Output : Down sampled tables $A' \subseteq A$ and $B' \subseteq B$

procedure DOWNSAMPLE(A, B, N, k, t)
  $W \leftarrow$ empty cluster map
  for $v \in A \cup B$ **do**
    $W[v] \leftarrow \{x\}$
  end for
  $C \leftarrow$ *SparklyAuto*(A, B, k)
  $C' \leftarrow$ *SortDescending*(C)
  for $(a, b) \in C'$ **do**
    if $|W[a]| + |W[b]| \leq t$ **then**
      $w \leftarrow W[a] \cup W[b]$
      $W[a] \leftarrow w$
      $W[b] \leftarrow w$
      if $|S| \geq N$ **then**
        break
      end if
    end if
  end for
  $A' \leftarrow \emptyset$
  $B' \leftarrow \emptyset$
  for $(a, b) \in C'$ **do**
    if $W[a] = W[b]$ **then**
      for $v \in W[a]$ **do**
        if $v \in A$ **then**
          $A' \leftarrow A' \cup \{v\}$
        else
          $B' \leftarrow B' \cup \{v\}$
        end if
      end for
    end if
  end for
  return A', B'
end procedure

Algorithm 4.3 Pseudocode for our upsampling procedure

Input : Tables A and B to be upsampled, gold matches $G \subseteq A \times B$, scale factor $q \in \mathbb{R}$

Output : Upsampled tables A' and B', and gold G'

procedure UPSAMPLE(A, B, G, p)

$A', B', G' \leftarrow$ GENERATEMATCHINGRECORDS(A, B, G, p)

$A'' \leftarrow A' \cup$ GENERATENONMATCHINGRECORDS($A, \lfloor |A| \cdot p \rfloor - |A'|$)

$B'' \leftarrow B' \cup$ GENERATENONMATCHINGRECORDS($B, \lfloor |B| \cdot p \rfloor - |B'|$)

return A'', B'', G'

end procedure

Algorithm 4.4 Pseudocode for generating records with matches

Input : Tables A and B, gold matches $G \subseteq A \times B$, scale factor $q \in \mathbb{R}$

Output : Upsampled tables A' and B', and gold $G' \subseteq A' \times B'$, $|G'| \approxeq |G| \cdot q$

procedure GENERATEMATCHINGRECORDS(A, B, G, p)

$G' \leftarrow G$

$A' \leftarrow \emptyset$

$B' \leftarrow \emptyset$

while $|G'| < |G| \cdot p$ **do**

$x \leftarrow$ RANDOMITEM($A \cup B$)

$x' \leftarrow$ PERTURBRECORD(x)

if $x \in A$ **then**

$G' \leftarrow G' \cup \{(x', b) | (a, b) \in G \wedge a = x\}$

$A' \leftarrow A' \cup \{x'\}$

else

$G' \leftarrow G' \cup \{(a, x') | (a, b) \in G \wedge b = x\}$

$B' \leftarrow B' \cup \{x'\}$

end if

end while

return $A \cup A', B \cup B', G'$

end procedure

Algorithm 4.5 Pseudocode for perturbing records to mimic common data quality issues

Input : A record x

Output : A perturbed record x'

procedure PERTURBRECORD(x)
  $F \leftarrow \{\text{CONCAT}, \text{TYPO}, \text{TRUNCATE}\}$
  $i \leftarrow \text{RANDOMINTEGER}(2, 3)$
  $x' \leftarrow x$
  while $i \neq 0$ **do**
    $f \leftarrow \text{RANDOMITEM}(F)$
    $x' \leftarrow f(x')$
    $i \leftarrow i - 1$
  end while
  return x'
end procedure

Algorithm 4.6 Pseudocode for generating non-matching records

Input : A table T, number of values to generate $N \in \mathbb{Z}^+$

Output : A new table T' with the same columns as T where $|T'| = N$

procedure GENERATENONMATCHINGRECORDS(T, N)
  $W \leftarrow ()$
  for $T_i \in \text{COLUMNS}(T)$ **do**
    if $\text{COLUMNTYPE}(T_i) =$ *numeric* **then**
      $T'_i \leftarrow \text{GENERATENUMERICCOLUMN}(T_i, N)$
    else if $\text{COLUMNTYPE}(T_i) =$ *string* **then**
      $T'_i \leftarrow \text{GENERATESTRINGCOLUMN}(T_i, N)$
    else
      error
    end if
    $W \leftarrow W \parallel \{T'_i\}$
  end for
  $T' \leftarrow \text{JOINCOLUMNS}(W)$
  return T'
end procedure

Algorithm 4.7 Pseudocode for generating string columns

Input : String column T_i, number of values to generate $N \in \mathbb{Z}^+$

Output : A new column of string values T_i' where $|T_i'| = N$

procedure GENERATESTRINGCOLUMN(T_i, N)

$S \leftarrow$ GENERATEVALUES($T_i, \min(|A_i|, 50000)$)

$P \leftarrow \{$TOKENIZE$(x) | x \in T_i \parallel S \wedge x \neq$ *null*$\}$

$M \leftarrow$ FITKDEMODEL($(|t| \ | t \in W)$)

$q \leftarrow \sum_{x \in T_i} \mathbb{1}(x =$ *null*$)/|T_i|$

$L_i \leftarrow$ SAMPLEVALUES(M, N)

$T_i' \leftarrow ()$

for $l \in L_i$ **do**

$W \leftarrow$ SAMPLETOKENS(P, l)

$s \leftarrow$ CONCATENATETOKENS(W)

$T_i' \leftarrow T_i' \parallel (s)$

end for

$T_i' \leftarrow$ INJECTNULLVALUES($T_i', \lfloor N \cdot q \rfloor$)

return T_i'

end procedure

Algorithm 4.8 Pseudocode for generating numeric columns

Input : Numeric column T_i, number of values to generate $N \in \mathbb{Z}^+$

Output : A new column of numeric values T_i' where $|T_i'| = N$

procedure GENERATENUMERICCOLUMN(T_i, N)

$M \leftarrow$ FITKDEMODEL(T_i)

$q \leftarrow \sum_{x \in T_i} \mathbb{1}(x =$ *null*$)/|T_i|$

$T_i' \leftarrow$ SAMPLEVALUES(M, N)

$T_i' \leftarrow$ INJECTNULLVALUES($T_i', \lfloor N \cdot q \rfloor$)

return T_i'

end procedure

Chapter 5

Conclusions

Entity matching (EM) remains one of the central challenges in data integration. As modern data systems grow in scale and heterogeneity, efficient and accurate blocking has become increasingly vital for enabling large-scale EM in practice. This dissertation has explored this problem in depth and introduced a set of novel contributions that advance the state of the art in both algorithmic design and empirical evaluation for blocking.

5.1 Summary of Contributions

This work made three primary contributions: Sparkly, a distributed TF–IDF-based blocker; Delex, a declarative and optimizable framework for composing blocking strategies; and BigGoat, a benchmark and data-generation suite for evaluating blocker scalability and accuracy under realistic conditions.

Sparkly demonstrated that classical information-retrieval techniques, when properly engineered for distributed settings, can deliver exceptional performance for EM. By building on Apache Spark and Lucene, Sparkly performs top-k blocking in a shared-nothing architecture that scales across cluster nodes. Its use of TF–IDF and BM25 scoring — combined with automated selection of blocking attributes and tokenizers — provides a flexible, data-driven approach that consistently achieved higher recall and smaller candidate sets than eight state-of-the-art baselines. Sparkly thus bridges a surprising gap between traditional IR theory and modern entity matching, showing that TF–IDF–based methods deserve renewed attention within the EM community.

Delex addressed a long-standing gap in the blocking ecosystem: the lack of a unified, declarative execution engine for composing multiple blocking methods. Delex introduced a simple but powerful language for expressing blocking programs, together with an optimizer that rewrites, estimates cost, and selects efficient execution plans automatically. The system supports predicate reuse, short-circuiting, and chunk-based execution to manage large data volumes. Empirical results showed that Delex can reduce runtime by up to threefold while maintaining high recall, and that it scales with both data size and cluster resources. This contribution lays a foundation for treating blocking not as ad hoc code but as an optimizable data-processing task akin to query planning in relational databases.

BigGoat complemented these algorithmic advances by filling a methodological void: the absence of a benchmark that captures the real-world scale and complexity of blocking workloads. BigGoat provides five large, diverse datasets — spanning citations, voter records, music databases, and the Web Data Commons — along with synthetic data generation via down- and up-sampling procedures. The down-sampling algorithm allows accurate recall estimation for blockers without running full quadratic comparisons, while the up-sampling framework offers a controlled means to explore scalability properties. Together, these tools provide the first publicly available platform for systematic evaluation of blockers at tens of millions of records.

5.2 Key Insights and Broader Lessons

Across these contributions, several broader insights emerge.

First, simplicity can outperform complexity. Sparkly's success illustrates that well-understood models such as TF–IDF, when re-contextualized within distributed architectures, can rival or surpass recent deep-learning approaches. The key lies not in ever-more-complex models but in thoughtful engineering that balances recall, candidate-set size, and scalability.

Second, declarative design enables composability and optimization. By treating blocking as a program that can be rewritten and costed, Delex opens the door to a new class of intelligent data-integration systems. Its optimizer brings transparency and reproducibility to a process that has long depended on manual tuning.

Third, evaluation at scale matters. Without realistic datasets, research in blocking risks overfitting to toy benchmarks that fail to expose the true challenges. BigGoat establishes an essential testing ground for future work, promoting scientific rigor and practical relevance.

Together, these insights suggest a broader methodological shift: blocking should be viewed not merely as a preprocessing step but as a core system component—one that can be formally specified, optimized, and evaluated like any other part of the data-management stack.

5.3 Future Directions

The work in this dissertation opens multiple avenues for continued exploration.

Learning-based configuration and adaptive blocking. Sparkly and Delex currently rely on heuristics for parameter selection (e.g., k values, tokenizer choice). Integrating automated learning mechanisms that tune these parameters from feedback data could yield adaptive blockers that improve over time.

Integration with end-to-end EM pipelines. Future systems could couple Delex's declarative blocking with downstream matching and fusion components, forming a unified query-optimization framework for entity resolution.

Hybrid semantic-syntactic blocking. Combining Sparkly's TF–IDF strengths with embedding-based similarity may capture both surface and semantic cues. Exploring how deep models can be used selectively within a scalable, Lucene-style architecture remains an open challenge.

Benchmark evolution. Expanding BigGoat with new domains (e.g., scientific data, e-commerce, or multilingual sources) and richer metadata would further strengthen its value to the community. Public leaderboards or reproducible evaluation pipelines could foster continued progress.

Human-in-the-loop and interactive blocking. Real-world data integration often requires domain feedback. Incorporating user interaction into Delex's declarative model could allow iterative refinement of blocking rules based on sampled errors or visual analytics.

5.4 Concluding Remarks

In summary, this dissertation has advanced the science and practice of blocking for entity matching through the design of scalable algorithms, declarative systems, and rigorous evaluation methodologies. The three systems — Sparkly, Delex, and BigGoat — form a cohesive framework that not only improves performance today but also redefines how researchers and practitioners conceptualize blocking.

By bringing together ideas from information retrieval, databases, and distributed systems, this work demonstrates that effective blocking can be both principled and practical. As data volumes continue to expand, the methods developed here will serve as building blocks for the next generation of large-scale entity-resolution systems—systems that are faster, more transparent, and more adaptable to the diverse data landscapes of the future.

Bibliography

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

[2] P. C. Arocena, B. Glavic, G. Mecca, R. J. Miller, P. Papotti, and D. Santoro. Messing up with bart: error generation for evaluating data-cleaning algorithms. *Proc. VLDB Endow.*, 9(2):36–47, Oct. 2015.

[3] N. Barlaug and J. A. Gulla. Neural networks for entity matching. *arXiv preprint arXiv:2010.11075*, 2020.

[4] J. L. Bentley and A. C.-C. Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–87, 1976.

[5] T. Bertin-Mahieux, D. P. Ellis, B. Whitman, and P. Lamere. The million song dataset. In *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*, 2011.

[6] A. Bonica. A data-driven voter guide for u.s. elections: Adapting quantitative measures of the preferences and priorities of political elites to help voters learn about candidates. *RSF: The Russell Sage Foundation Journal of the Social Sciences*, 2(7):11–32, 2016.

[7] A. Borthwick, S. Ash, B. Pang, S. Qureshi, and T. Jones. Scalable blocking for very large databases. In I. Koprinska, M. Kamp, A. Appice, C. Loglisci, L. Antonie, A. Zimmermann, R. Guidotti, Ö. Özgöbek, R. P. Ribeiro, R. Gavaldà, J. Gama, L. Adilova, Y. Krishnamurthy, P. M. Ferreira, D. Malerba, I. Medeiros, M. Ceci, G. Manco, E. Masciari, Z. W. Ras, P. Christen, E. Ntoutsi, E. Schubert, A. Zimek, A. Monreale, P. Biecek, S. Rinzivillo, B. Kille, A. Lommatzsch, and J. A. Gulla, editors, *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*, volume 1323 of *Communications in Computer and Information Science*, pages 303–319. Springer, 2020.

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

[9] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In A. Y. Halevy, Z. G. Ives, and A. Doan, editors, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 313–324. ACM, 2003.

[10] P. Christen. Development and user experiences of an open source data cleaning, deduplication and record linkage system. *SIGKDD Explor. Newsl.*, 11(1):39–48, Nov. 2009.

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

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

[13] P. Christen and D. Vatsalan. Flexible and extensible generation and corruption of personal data. In *Proceedings of the 22nd ACM International Conference on Information & Knowledge Management*, CIKM '13, page 1165–1168, New York, NY, USA, 2013. Association for Computing Machinery.

[14] V. Christophides, V. Efthymiou, T. Palpanas, G. Papadakis, and K. Stefanidis. An overview of end-to-end entity resolution for big data. *ACM Computing Surveys (CSUR)*, 53(6):1–42, 2020.

[15] W. W. Cohen, P. Ravikumar, and S. E. Fienberg. A comparison of string distance metrics for name-matching tasks. In S. Kambhampati and C. A. Knoblock, editors, *Proceedings of IJCAI-03 Workshop on Information Integration on the Web (IIWeb-03), August 9-10, 2003, Acapulco, Mexico*, pages 73–78, 2003.

[16] F. E. Commission. FEC contribution data.

[17] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[18] V. Crescenzi, A. D. Angelis, D. Firmani, M. Mazzei, P. Merialdo, F. Piai, and D. Srivastava. Alaska: A flexible benchmark for data integration tasks. *ArXiv*, abs/2101.11259, 2021.

[19] S. Das, P. S. G. C., A. Doan, J. F. Naughton, G. Krishnan, R. Deep, E. Arcaute, V. Raghavendra, and Y. Park. Falcon: Scaling up hands-off crowdsourced entity matching to build cloud services. In *SIGMOD*, 2017.

[20] S. Das et al. The magellan data repository. `https://sites.google.com/site/anhaidgroup/useful-stuff/data`.

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

[22] dblp Team. dblp computer science bibliography – Monthly Snapshot XML Release.

[23] C. Dimopoulos, S. Nepomnyachiy, and T. Suel. Optimizing top-k document retrieval strategies for block-max indexes. In S. Leonardi, A. Panconesi, P. Ferragina, and A. Gionis, editors, *Sixth ACM International Conference on Web Search and Data Mining, WSDM 2013, Rome, Italy, February 4-8, 2013*, pages 113–122. ACM, 2013.

[24] S. Ding and T. Suel. Faster top-k document retrieval using block-max indexes. In W. Ma, J. Nie, R. Baeza-Yates, T. Chua, and W. B. Croft, editors, *Proceeding of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2011, Beijing, China, July 25-29, 2011*, pages 993–1002. ACM, 2011.

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

[26] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *TKDE*, 19(1), 2007.

[27] C. C. Foundation. Common Crawl Dataset.

[28] C. Gokhale, S. Das, A. Doan, J. F. Naughton, N. Rampalli, J. Shavlik, and X. Zhu. Corleone: hands-off crowdsourcing for entity matching. SIGMOD, 2014.

[29] Y. Govind et al. Cloudmatcher: A cloud/crowd service for entity matching. BIGDAS@KDD, 2017.

[30] Y. Govind, P. Konda, P. S. G. C., P. Martinkus, P. Nagarajan, H. Li, A. Soundararajan, S. Mudgal, J. R. Ballard, H. Zhang, A. Ardalan, S. Das, D. Paulsen, A. S. Saini, E. Paulson, Y. Park, M. Carter, M. Sun, G. M. Fung, and A. Doan. Entity matching meets data science: A progress report from the magellan project. In P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 389–403. ACM, 2019.

[31] Y. Govind, E. Paulson, P. Nagarajan, P. S. G. C., A. Doan, Y. Park, G. Fung, D. Conathan, M. Carter, and M. Sun. Cloudmatcher: A hands-off cloud/crowd service for entity matching. *Proc. VLDB Endow.*, 11(12):2042–2045, 2018.

[32] A. Grand, R. Muir, J. Ferenczi, and J. Lin. From MAXSCORE to block-max wand: The story of how lucene significantly improved query evaluation performance. In J. M. Jose, E. Yilmaz, J. Magalhães, P. Castells, N. Ferro, M. J. Silva, and F. Martins, editors, *Advances in Information Retrieval - 42nd European Conference on IR Research, ECIR 2020, Lisbon, Portugal, April 14-17, 2020, Proceedings, Part II*, volume 12036 of *Lecture Notes in Computer Science*, pages 20–27. Springer, 2020.

[33] M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. *SIGMOD Rec.*, 24(2):127–138, May 1995.

[34] M. Hernández and S. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Min. Knowl. Discov.*, 2:9–37, 01 1998.

[35] K. Hildebrandt, F. Panse, N. Wilcke, and N. Ritter. Large-scale data pollution with apache spark. *IEEE Transactions on Big Data*, 6(2):396–411, 2020.

[36] E. Ioannou and Y. Velegrakis. Embench++: Data for a thorough benchmarking of matching-related methods: Homepage→ https://db.disi.unitn.eu/pages/embench/. *Semantic Web*, 10(2):435–450, 2019.

[37] P. Konda, S. Das, P. Suganthan GC, A. Doan, A. Ardalan, J. R. Ballard, H. Li, F. Panahi, H. Zhang, J. Naughton, et al. Magellan: Toward building entity matching management systems. *PVLDB*, 9(13):1581–1584, 2016.

[38] H. Köpcke, A. Thor, and E. Rahm. Evaluation of entity resolution approaches on real-world match problems. VLDB, 2010.

[39] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.

[40] P. Koutris, S. Salihoglu, and D. Suciu. Algorithmic aspects of parallel data processing. *Found. Trends Databases*, 8(4):239–370, 2018.

[41] H. Li, P. Konda, P. S. GC, A. Doan, B. Snyder, Y. Park, G. Krishnan, R. Deep, and V. Raghavendra. Matchcatcher: A debugger for blocking in entity matching. In *EDBT*, pages 193–204, 2018.

[42] P. Li, X. Cheng, X. Chu, Y. He, and S. Chaudhuri. Auto-fuzzyjoin: Auto-program fuzzy similarity joins without labeled examples. In G. Li, Z. Li, S. Idreos, and D. Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 1064–1076. ACM, 2021.

[43] Y. Li, J. Li, Y. Suhara, A. Doan, and W.-C. Tan. Deep entity matching with pre-trained language models. *PVLDB*, 14(1):50–60, 2020.

[44] F. Lin and W. W. Cohen. Power iteration clustering. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, page 655–662, Madison, WI, USA, 2010. Omnipress.

[45] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*. Cambridge University Press, 2008.

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

[47] S. Mudgal, H. Li, T. Rekatsinas, A. Doan, Y. Park, G. Krishnan, R. Deep, E. Arcaute, and V. Raghavendra. Deep learning for entity matching: A design space exploration. In *SIGMOD*, 2018.

[48] F. Naumann and M. Herschel. An introduction to duplicate detection. *Synthesis Lectures on Data Management*, 2(1):1–87, 2010.

[49] K. O'Hare, A. Jurek-Loughrey, and C. P. de Campos. High-value token-blocking: Efficient blocking method for record linkage. *ACM Trans. Knowl. Discov. Data*, 16(2):24:1–24:17, 2022.

[50] K. O'Hare, A. Jurek-Loughrey, and C. de Campos. A review of unsupervised and semi-supervised blocking methods for record linkage. *Linking and Mining Heterogeneous and Multi-view Data*, pages 79–105, 2019.

[51] F. Panse, W. Wingerath, and B. Wollmer. Towards scalable generation of realistic test data for duplicate detection, 2023.

[52] G. Papadakis, G. Alexiou, et al. Schema-agnostic vs schema-based configurations for blocking methods on homogeneous data. VLDB, 2015.

[53] G. Papadakis, E. Ioannou, E. Thanos, and T. Palpanas. The four generations of entity resolution. *Synthesis Lectures on Data Management*, 16(2):1–170, 2021.

[54] G. Papadakis, G. Mandilaras, L. Gagliardelli, G. Simonini, E. Thanos, G. Giannakopoulos, S. Bergamaschi, T. Palpanas, and M. Koubarakis. Three-dimensional entity resolution with jedai. *Information Systems*, 93:101565, 2020.

[55] G. Papadakis, D. Skoutas, E. Thanos, and T. Palpanas. Blocking and filtering techniques for entity resolution: A survey. *ACM Computing Surveys (CSUR)*, 53(2):1–42, 2020.

[56] G. Papadakis, L. Tsekouras, E. Thanos, N. Pittaras, G. Simonini, D. Skoutas, P. Isaris, G. Giannakopoulos, T. Palpanas, and M. Koubarakis. Jedai3: beyond batch, blocking-based entity resolution. In *EDBT*, pages 603–606, 2020.

[57] D. Paulsen, Y. Govind, and A. Doan. Homepage of the sparkly blocking system. Technical report, 2022. `http://pages.cs.wisc.edu/~anhai/sparkly.html`.

[58] G. Ppadakis, M. Fisichella, F. Schoger, G. Mandilaras, N. Augsten, and W. Nejdl. Benchmarking filtering techniques for entity resolution. Technical report, 2022. arXiv:2022.12521v3.

[59] A. Primpeli, R. Peeters, and C. Bizer. The WDC training dataset and gold standard for large-scale product matching. In S. Amer-Yahia, M. Mahdian, A. Goel, G. Houben, K. Lerman, J. J. McAuley, R. Baeza-Yates, and L. Zia, editors, *Companion of The 2019 World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, pages 381–386. ACM, 2019.

[60] R. Seitz. Understanding tf/idf and bm25. Technical report, 2022. `https://kmwllc.com/index.php/2020/03/20/understanding-tf-idf-and-bm-25`.

[61] M. Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[62] S. Thirumuruganathan, H. Li, N. Tang, M. Ouzzani, Y. Govind, D. Paulsen, G. Fung, and A. Doan. Deep learning for blocking in entity matching: A design space exploration. *Proc. VLDB Endow.*, 14(11):2459–2472, 2021.

[63] J. Wang, Y. Li, and W. Hirota. Machamp: A generalized entity matching benchmark. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, CIKM '21, page 4633–4642, New York, NY, USA, 2021. Association for Computing Machinery.

[64] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

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

[66] M. Yu, G. Li, D. Deng, and J. Feng. String similarity search and join: a survey. *Frontiers Comput. Sci.*, 10(3):399–417, 2016.

[67] W. Zhang, H. Wei, B. Sisman, X. L. Dong, C. Faloutsos, and D. Page. Autoblock: A hands-off blocking framework for entity matching. In *WSDM*, pages 744–752, 2020.

APPENDIX
Delex Blocking Programs

For the evaluation of Delex we use the following blocking programs.

A.1 DBLP program 2

```
KEEP (
    (
    BM25_topk(standard, title, title, 20)
     AND
    overlap_coeff(alnum_tokens, authors, authors) >= 0.6
    )
  OR
    (
    BM25_topk(standard, title, title, 20)
     AND
    cosine(3gram_tokens, authors, authors) >= 0.7
    )
  OR
    (
    BM25_topk(standard, authors, authors, 25)
     AND
    overlap_coeff(stripped_whitespace_tokens, title, title) >= 0.8
```

```
    )
)
DROP (
    (
    edit_distance(year, year) <= 0.7
    )
)
```

A.2 DBLP program 1

```
KEEP (
    (
    jaccard(3gram_tokens, title, title) >= 0.8
     AND
    overlap_coeff(alnum_tokens, authors, authors) >= 0.6
    )
  OR
    (
    overlap_coeff(stripped_whitespace_tokens, title, title) >= 0.9
     AND
    jaccard(3gram_tokens, authors, authors) >= 0.8
    )
  OR
    (
    smith_waterman[1.0](title, title) >= 0.9
     AND
    jaccard(3gram_tokens, authors, authors) >= 0.8
    )
  OR
```

```
    (
    jaccard(3gram_tokens, title, title) >= 0.8
     AND
    jaccard(alnum_tokens, authors, authors) >= 0.8
    )
)
DROP (
    (
    edit_distance(year, year) <= 0.7
    )
)
```

A.3 FEC January 2022 program 1

```
KEEP (
    (
    name_match(contributor_name, contributor_name) == True
     AND
    exact_match(contributor_zipcode, contributor_zipcode) == True
    )
  OR
    (
    name_match(contributor_name, contributor_name) == True
     AND
    lowercase_exact_match(contributor_city, contributor_city) == True
    )
  OR
    (
    name_match(contributor_name, contributor_name) == True
```

```
     AND
    edit_distance(contributor_zipcode, contributor_zipcode) >= 0.8
     AND
    edit_distance(contributor_city, contributor_city) >= 0.8
    )
)
DROP (
    (
    jaccard(3gram_tokens, contributor_name, contributor_name) <= 0.7
    )
  OR
    (
    lowercase_exact_match(contributor_state, contributor_state) == False
    )
)
```

A.4 FEC January 2022 program 2

```
KEEP (
    (
    BM25_topk(3gram, contributor_name, contributor_name, 25)
     AND
    jaro_winkler[0.1](contributor_zipcode, contributor_zipcode) >= 0.8
     AND
    smith_waterman[1.0](contributor_city, contributor_city) >= 0.8
    )
  OR
    (
    BM25_topk(3gram, contributor_name, contributor_name, 25)
```

```
     AND
    exact_match(contributor_zipcode, contributor_zipcode) == True
    )
  OR
    (
    BM25_topk(3gram, contributor_name, contributor_name, 25)
     AND
    lowercase_exact_match(contributor_city, contributor_city) == True
    )
)
DROP (
    (
    jaccard(3gram_tokens, contributor_name, contributor_name) <= 0.7
    )
  OR
    (
    lowercase_exact_match(contributor_state, contributor_state) == False
    )
)
```

A.5 FEC

```
KEEP (
    (
    BM25_topk(standard, contributor_name, contributor_name, 250)
     AND
    jaro_winkler[0.1](contributor_zipcode, contributor_zipcode) >= 0.8
     AND
    smith_waterman[1.0](contributor_city, contributor_city) >= 0.8
```

```
    )
  OR
    (
    name_match(contributor_name, contributor_name) == True
     AND
    jaro_winkler[0.1](contributor_zipcode, contributor_zipcode) >= 0.8
    )
  OR
    (
    BM25_topk(standard, contributor_name, contributor_name, 250)
     AND
    lowercase_exact_match(contributor_city, contributor_city) == True
    )
)
DROP (
    (
    cosine(3gram_tokens, contributor_name, contributor_name) <= 0.7
    )
  OR
    (
    overlap_coeff(alnum_tokens, contributor_name, contributor_name) <= 0.5
    )
  OR
    (
    lowercase_exact_match(contributor_state, contributor_state) == False
    )
)
```

A.6 WDC program 1

```
KEEP (
    (
    jaccard(alnum_tokens, name, name) >= 0.8
    )
  OR
    (
    cosine(3gram_tokens, name, name) >= 0.9
     AND
    exact_match(gtin13, gtin13) == True
    )
  OR
    (
    jaccard(alnum_tokens, description, description) >= 0.9
    )
)
DROP (
    (
    jaro_winkler[0.1](brand, brand) <= 0.5
     AND
    overlap_coeff(alnum_tokens, description, description) <= 0.5
    )
  OR
    (
    overlap_coeff(stripped_whitespace_tokens, name, name) <= 0.5
    )
)
```

A.7 WDC program 2

```
KEEP (
    (
    BM25_topk(standard_stopwords, name, name, 25)
     AND
    jaro_winkler[0.1](brand, brand) >= 0.5
    )
  OR
    (
    BM25_topk(standard_stopwords, name, name, 25)
     AND
    overlap_coeff(alnum_tokens, description, description) >= 0.8
    )
)
```

A.8 Music program 2

```
KEEP (
    (
    BM25_topk(3gram, title, title, 25)
    )
  OR
    (
    jaccard(stripped_whitespace_tokens, title, title) >= 0.7
    )
)
DROP (
    (
```

```
    jaccard(3gram_tokens, title, title) < 0.4
     AND
    lowercase_exact_match(title, title) == False
    )
  OR
    (
    jaccard(3gram_tokens, artist_name, artist_name) < 0.4
     AND
    lowercase_exact_match(artist_name, artist_name) == False
    )
)
```

A.9 Music program 1

```
KEEP (
    (
    cosine(stripped_whitespace_tokens, title, title) >= 0.7
     AND
    jaccard(3gram_tokens, artist_name, artist_name) >= 0.7
    )
  OR
    (
    cosine(stripped_whitespace_tokens, title, title) >= 0.7
     AND
    jaccard(3gram_tokens, release, release) >= 0.7
    )
  OR
    (
    cosine(stripped_whitespace_tokens, artist_name, artist_name) >= 0.7
```

```
     AND
    jaccard(3gram_tokens, title, title) >= 0.7
    )
  OR
    (
    cosine(stripped_whitespace_tokens, release, release) >= 0.7
     AND
    jaccard(3gram_tokens, title, title) >= 0.7
    )
  OR
    (
    cosine(stripped_whitespace_tokens, release, release) >= 0.7
     AND
    jaccard(3gram_tokens, artist_name, artist_name) >= 0.7
    )
)
DROP (
    (
    jaccard(3gram_tokens, title, title) < 0.5
     AND
    lowercase_exact_match(title, title) == False
    )
)
```

A.10 Music Brainz program 1

```
KEEP (
    (
    cosine(stripped_whitespace_tokens, title, title) >= 0.7
```

```
    AND
    jaccard(3gram_tokens, artist, artist) >= 0.7
    )
  OR
    (
    cosine(stripped_whitespace_tokens, title, title) >= 0.7
    AND
    jaccard(3gram_tokens, album, album) >= 0.7
    )
  OR
    (
    jaccard(3gram_tokens, title, title) >= 0.7
    AND
    cosine(stripped_whitespace_tokens, artist, artist) >= 0.7
    )
  OR
    (
    jaccard(3gram_tokens, title, title) >= 0.7
    AND
    cosine(stripped_whitespace_tokens, album, album) >= 0.7
    )
  OR
    (
    jaccard(3gram_tokens, artist, artist) >= 0.7
    AND
    cosine(stripped_whitespace_tokens, album, album) >= 0.7
    )
)
```

```
DROP (
    (
    jaccard(3gram_tokens, title, title) < 0.5
     AND
    lowercase_exact_match(title, title) == False
    )
)
```

A.11 Music Brainz program 2

```
KEEP (
    (
    BM25_topk(standard, title, title, 50)
     AND
    lowercase_exact_match(album, album) == True
    )
  OR
    (
    BM25_topk(standard, title, title, 50)
     AND
    cosine(3gram_tokens, artist, artist) >= 0.8
    )
  OR
    (
    BM25_topk(standard, title, title, 50)
     AND
    overlap_coeff(stripped_whitespace_tokens, artist, artist) >= 0.9
    )
  OR
```

```
    (
    jaccard(3gram_tokens, title, title) >= 0.8
     AND
    lowercase_exact_match(album, album) == True
    )
)
DROP (
    (
    jaccard(3gram_tokens, title, title) < 0.5
    )
  OR
    (
    jaccard(3gram_tokens, artist, artist) < 0.5
    )
)
```