JOSIE: Overlap Set Similarity Search for Finding Joinable Tables in Data Lakes

Erkang Zhu
University of Toronto
Toronto, Canada
ekzhu@cs.toronto.edu

Dong Deng
Rutgers University, USA &
Inception Institute of Artificial Intelligence, UAE
dong.deng@rutgers.edu

Fatemeh Nargesian
University of Toronto
Toronto, Canada
fnargesian@cs.toronto.edu

Renée J. Miller
Northeastern University
Boston, USA
miller@northeastern.edu

ABSTRACT

We present a new solution for finding joinable tables in massive data lakes: given a table and one join column, find tables that can be joined with the given table on the largest number of distinct values. The problem can be formulated as an overlap set similarity search problem by considering columns as sets and matching values as intersection between sets. Although set similarity search is well-studied in the field of approximate string search (e.g., fuzzy keyword search), the solutions are designed for and evaluated over sets of relatively small size (average set size rarely much over 100 and maximum set size in the low thousands) with modest dictionary sizes (the total number of distinct values in all sets is only a few million). We observe that modern data lakes typically have massive set sizes (with maximum set sizes that may be tens of millions) and dictionaries that include hundreds of millions of distinct values. Our new algorithm, JOSIE (**JO**ining **S**earch using **I**ntersection **E**stimation) minimizes the cost of set reads and inverted index probes used in finding the top-k sets. We show that JOSIE completely out performs the state-of-the-art overlap set similarity search techniques on data lakes. More surprising, we also consider state-of-the-art approximate algorithm and show that our new exact search algorithm performs almost as well, and even in some cases better, on real data lakes.

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3300065

ACM Reference Format:
Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée J. Miller. 2019. JOSIE: Overlap Set Similarity Search for Finding Joinable Tables in Data Lakes . In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3299869.3300065

1 INTRODUCTION

Set similarity search has numerous applications ranging from document search and keyword queries to entity-identification and data cleaning [33]. Overlap set similarity search is an instance of this problem where the similarity measure used is the intersection size[1] of the sets. Unlike other similarity measures (like Jaccard or Cosine), set intersection size is not biased toward small sets [25]. Recently, overlap set similarity search was used to find joinable tables in a data lake [34][2]. We illustrate this application with an example.

Example 1. *A user inputs a query table T_Q and specifies a column Q as the join column along with a threshold θ. The* **join table search problem** *[34] is to find all tables T_X with a column X such that $|Q \cap X| \geq \theta$ meaning the intersection size between X and Q is high. Table 1:(a) is an example query table about air pollution emissions of industrial facilities in Canada[3]. If the user sets* `Postal` *column as Q, we can find a candidate table on political campaign contributions* [4] *with a* `Postal Code` *column (X) that contains Q. By joining these two tables, we can produce an interesting analysis on pollution and political orientation.*

[1] Past work in set similarity search sometimes used "overlap similarity" [29, 32], while "intersection size" is less ambiguous. In this paper we use the latter, but keep the common name "overlap set similarity search".
[2] Zhu et al. [34] actually use containment ($|Q \cap X|)/|Q| \geq \theta$, but containment is easily converted to intersection size.
[3] https://open.canada.ca/data/en/dataset/f6660f68-5e78-47b3-9306-9805e1d3d6e9
[4] https://open.canada.ca/data/en/dataset/ef1e3528-b570-4a42-92ef-18a9749af8f2

(a) Example query table

Facility	Comapny	Postal	NOx
Oil Sands	Suncor Energy	T9H3E3	9340.81
Carberry Factory	McCain Foods	R0K0H0	50.7
Grand Falls	McCain Foods	E3Y4A5	60.6
Ingleside	Kraft Heinz	K0C1M0	-99

(b) Example candidate table

Contributor	Postal Code	Recipient Party	Amount
E*,B*	T9H3E3	Liberal	397.46
J*,S*	R0K0H0	Conservative	500.00
J*,S*	R0K0H0	Conservative	250.00
M*,S*	K0C1M0	Liberal	400.00

Table 1: Example joinable tables

Notice that the set intersection size is the number of tuples in T_Q that join with T_X and hence is a good measure of equi-joinability. It is immune to data value skew in Q and aggregation operations that may be needed before or after performing the join. For columns of floating point numbers or long text, set intersection size may not be the right choice, however, since the user must pick Q, the choice is on the user, who should understand the semantic of equi-join. Finding joinable tables can help data scientists to discover useful tables that they may not know about, and may even lead to surprising new data science.

Solutions for overlap set similarity search have focused on relatively small sets such as keywords and titles [3, 9, 16, 28, 29, 31]. These approaches work well for applications like similarity joins (where the sets are n-grams of a single attribute value) or entity-resolution (where the sets are an entity-name or tuple). These state-of-the-art solutions rely on *inverted indexes*, which contain *posting lists* that map every distinct value in a set (or token) to a list of indexed sets that contain it. The solutions involve either 1) reading all posting lists in the query to find candidates, and then ranking them based on number of times they appear, or 2) reading a subset of posting lists to locate candidates, and then reading the candidates to find the final ranked list of sets. Most of these solutions assume the inverted index is relatively small.

The use of overlap set similarity search for finding joinable tables and the characteristics of modern data lakes take this classic problem to the next level. Data lakes may have large set sizes and huge dictionaries (number of distinct values) as shown in Table 2. In this table, we include three data lakes. The first is a lake of Open Data (obtained from Nargesian et al. [21]). The second is from the WDC Web Table Corpus [14].

Table 2: Characteristics of sets derived from 215,393 Open Data tables in comparison with other datasets used in previous works. For Open Data and Web Tables, all numerical values are removed, as explained in Section 4.

	#Sets	MaxSize	AvgSize	#UniqTokens
OpenData	745,414	22,075,531	1,540	562,320,456
WebTable	163,510,917	17,030	10	184,644,583
Enterprise	2,032	859,765	4,011	3,902,604
AOL	10,054,184	245	3	3,900,000
ENRON	517,422	3,162	135	1,100,000
DBLP	100,000	1,625	86	6,864

The third are statistics from a private enterprise data lake from a large organization (obtained from Deng et al. [7]). The last three lines report not data lakes, but rather typical datasets used in previous work for set similarity search [3, 29, 31] (numbers from Mann et al. [18]). Notice that the previous work has used data with much smaller set sizes and far fewer unique values.

These differences have significant implications. First, due to the huge number of unique tokens, an inverted index will use a lot of space – approximately 100 GB in PostgreSQL. So, memory management becomes an issue for the index and reading the whole index (all posting lists) becomes infeasible. Second, sets are large as well so again memory management is an issue and it is important to limit the number of sets read. Consequently, the cost of reading a posting list or a set in data lakes will be more expensive than considered in previous work, and an approach that reads all the posting lists of a large query, or reads too many candidates, will not be feasible.

So far we have been considering *exact* solutions to overlap set similarity search. Approximate algorithms have also been studied [25, 34]. These algorithms are scalable in performance, however they tend to suffer from false positive and negative errors, especially when the distribution of set sizes is skewed, for example, following a Zipfian distribution. Zhu et al. [34] showed that attribute sizes in both Open Data and WebTables follow Zipfian distributions. One of our contributions is to show that our new exact approach can have just as good performance.

Previous work on set similarity search focuses on a version of the problem that involves a threshold: given a set similarity function that takes two sets as input and outputs a score, for a query, find all sets whose scores, computed with the query, meet a user-specified threshold [3, 16, 29]. Overlap set similarity can be defined as a similarity function that outputs the set intersection size. Zhu et al. [34] find all tables with a column X whose intersection size with the query column

Q is greater than a threshold. However, for the problem of searching data lakes, using a threshold may confuse users, who have no knowledge of what data exists in the lake and therefore do not know what is a good threshold that will retrieve some, but not too many answers. A threshold that is too low may severely delay response time as there are too many results to process, potentially causing the user to get frustrated.

An alternative version of this problem is top-k search: find the best k sets that have the largest intersection with Q [31].. The user does not need any prior knowledge to specify k. For example, a small value (e.g., 10) may be sufficient to get an understanding for what is available in the data lake. Xiao et al. [31] proposed a top-k algorithm that can be used for top-k overlap set similarity search (that we call *ProbeSet*), but it has never been evaluated over large sets.

Our contributions include the following.

(1) A novel exact top-k overlap set similarity search algorithm, JOSIE that scales to large sets and large dictionary sizes (including data lakes), making it a solution to the joinable table search problem.
(2) We show that our solution out-performs the state-of-the-art solutions in query time by a factor of two, with 3× less standard deviation, on two real-world data lakes: Government Open Data and Web Tables.
(3) Unlike the state-of-the-art solutions that are very sensitive to data characteristics (and hence perform very differently on different data lakes), our solution is adaptive to the data distribution and hence has robust performance even on data lakes as diverse as Open Data and Web Tables.
(4) Surprisingly, for reasonable k, our solution out-performs the state-of-the-art approximate solution (which uses approximate data sketches for sets) for query sizes up to 10K. Ours is the first experimental comparison of exact approaches to even consider queries of this size and being able to beat approximate approaches is a dramatic improvement in the field.
(5) We study massive queries up to 100K values and show that we are competitive with approximate techniques for small k. For larger k our performance is 3 to 4 times slower than approximate search (meaning 6-8 seconds instead of 1 or 2), but with this extra time we find exact results and the approximate technique can be missing from 10%-40% of the answers.

2 PRELIMINARIES

In this section we present a formal definition of the top-k overlap set similarity search problem, and some background material on inverted indexes, top-k, and canonicalization of tokens using a global order.

2.1 Problem Definition

A search engine for joinable tables asks a user to input a table and specify a column, and returns tables that can be joined with the input table on the specified column. The number of distinct values in the resulting join column is used to measure the relevance of the returned tables.

This problem can be expressed as a top-k overlap set similarity search problem. We take all columns of all tables in a repository of tables, convert every column into a set of distinct values, and call the big collection of sets Ω. Let Q be the set of distinct values in the user specified column. The top-k overlap set similarity search problem can be defined as follow:

Definition 1. *Top-k overlap set similarity search: given a set Q (the query), and a collection of sets Ω, find a sub-collection ω of at most k sets such that:*

(1) $|Q \cap X| > 0,\ X \in \omega$, *and*
(2) $\min\{|Q \cap X|,\ X \in \omega\} \geq |Q \cap Y|,\ Y \notin \omega,\ Y \in \Omega$

The key here is that the k-th (or the last) set in the result ranked by the size of intersection with the query, has the same or larger intersection size as any other set that is not in the result. As we will explain in the next section, the k-th intersection size in the result can be used as a threshold to prune unseen sets.

2.2 Inverted Index and Dictionary

Here, we describe inverted indexes in the context of overlap set similarity search. For a collection of sets $X_1, X_2, \ldots \in \Omega$, we extract the *tokens* (i.e., values) $x_1, x_2, \ldots$ from all the sets, and for each token, we build a *posting list* of pointers to the sets that contain the token, and these posting lists together form an inverted index.

An inverted index over a massive collection of sets with millions of tokens and millions of sets is very large, so the posting lists may have to be stored on disk or distributed as scalability becomes an issue. A query set may contain tokens not in the inverted index, and we need to determine this quickly. Thus, solutions typically store a data structure called a *dictionary*, that contains each token, its frequency and a pointer to its posting list.

Using the dictionary and inverted index there is a simple algorithm, that we call `MergeList`, for top-k overlap set similarity search [19]. First, initialize a hash map to store the intersecting token counts of *candidates* – sets discovered from posting lists. For every query token, use the dictionary to check if the token exists in the inverted index; if exists then read the entire posting list and increment counts for candidates in the list. Once all posting lists are read, we can sort the candidates by their intersecting token counts, and return the k sets with highest counts. The total time (ignoring

memory hierarchy concerns) is

$$\sum_{x_i \in Q \cap U} L(f_i) \tag{1}$$

where U is the set of all tokens in the dictionary, f_i is the frequency of token x_i, and $L(\cdot)$ is the time to read a posting list as a function of the length of the list.

`MergeList`'s read time is linear to the number of matched tokens. This is not a big issue when sets have less than a few hundred tokens, however, in the context of joinable table search, the sets extracted from columns can easily have thousands or even millions of tokens, as evidenced by Table 2. Thus, a better approach could be to reduce the number of posting lists read.

2.3 Prefix Filter

Prefix filter, an idea proposed by Chaudhuri et al., to solve the threshold version of set similarity search problem (see Section 1) [6]. The main idea is that given an intersection size threshold t, all candidates X such that $|Q \cap X| \geq t$ must be found in any subset (*prefix*) of posting lists given the subset's size is $|Q| - t + 1$, and posting lists outside the subset are not required to be read.

Xiao et al. proposed a top-k algorithm that uses prefix filter. It uses a fixed-size min-heap to keep track of the running top-k candidates [31]. The main trick of the algorithm is to use the running k-th candidate's intersection size as the threshold: after finishing reading a posting list, we need to check the intersection size of the current k-th candidate X_k, $|Q \cap X_k|$, and if the number of lists already read so far is equal to $|Q| - |Q \cap X_k| + 1$, then the algorithm stops reading new lists and returns the current running top-k candidates as the results. Since $|Q \cap X_k|$ is the threshold, the first $|Q| - |Q \cap X_k| + 1$ posting lists become the prefix. We call this algorithm `ProbeSet`, because as opposed to `MergeList` that reads only posting lists, it probes candidates as it encounters them.

2.4 Token Ordering and Position Filter

As described in the last section, `ProbeSet` reads and computes exact intersection size for every new candidate encountered. Xiao et al. introduced an optimization techinique called *position filter* that prunes out candidates whose intersection sizes are less than a threshold before reading them [32]. For the rest of this paper, `ProbeSet` always uses the position filter.

There are two requirements to use position filter. First, we must assign a global ordering (e.g., lexicalgraphic, length, etc.) for the universe of all tokens, and all sets (including the query set) must be sorted by the global ordering. The second requirement is that each token's posting list must contain the positions of the token in the sets that contain it, as well as the sizes of the sets.

Example 2. *An example of sets and posting lists.*

$$\begin{aligned} X_1 &= \{x_1, x_{100}, x_{200}\} \\ X_2 &= \{x_2, x_5\} \\ X_3 &= \{x_2\} \\ X_4 &= \{x_2, \ldots, x_{100}, x_{101}\} \end{aligned} \quad \Longrightarrow \quad \begin{aligned} x_1 &: \{(X_1, 1, 3)\} \\ x_2 &: \{(X_2, 1, 2), (X_3, 1, 1), \\ &\quad (X_4, 1, 100)\} \\ x_{100} &: \{(X_1, 2, 3), (X_4, 99, 100)\} \\ x_{200} &: \{(X_1, 3, 3)\} \end{aligned}$$

In each entry of a posting list, the second integer is the position, and the third integer is the size of the set.

When we encounter a new candidate X from the posting list of token x_i (i.e., the i-th token in the sorted query set), we can compute the upper-bound of its intersection size with the query set Q using the equation:

$$|Q \cap X| \leq |Q \cap X|_{ub} = 1 + \min\left(|Q| - i, |X| - j_{X,0}\right) \tag{2}$$

Where $j_{X,0}$ is the position of token x_i in X, it is also the first intersecting token between Q and X, hence the 0 in the subscript. If $|Q \cap X|_{ub} \leq |Q \cap X_k|$, we can skip X, and ignore it in future encounters.

Another benefit of using the position filter is reducing the time of reading individual candidate. Since there is no intersecting token between Q and X before their first matching positions, we only need to read the suffix $X[j_{X,0} + 1 :]$ to compute the exact intersection size.

Example 3. *Consider Example 2's posting lists and let the query set be $Q = \{x_1, x_2, x_{100}, x_{200}\}$ and $k = 2$. We first read the posting list of x_1, which leads us to read and compute the exact intersection size for X_1 which is three. Then we read the posting list of x_2. Since the heap is not full, we read X_2, compute the exact intersection size, and push $(X_2, 1)$ to the heap. The running heap is $\{(X_1, 3), (X_2, 1)\}$ and the k-th candidate's intersection size is one.*

Now the heap is full and we have running top-k, we can use the position filter to check the next candidate X_3. For X_3, because $|Q \cap X_3|_{ub} = 1 + \min(4 - 2, 1 - 1) = 1 \leq 1$, it does not pass the position filter. So we skip X_3. For X_4, because $|Q \cap X_4|_{ub} = 1 + \min(4 - 2, 100 - 1) = 100 > 1$, it passes the position filter. So we read X_4, and compute the exact intersection size which is – pop and push the heap, which is now $\{(X_1, 3), (X_4, 2)\}$.

We continue to read the third posting list x_{100}, and we can skip both X_1 and X_4 as we have seen them before. The prefix size is $4 - 2 + 1 = 3$, thus x_{100} is the last posting list. We skip x_{200} and terminate search.

The total read time of the optimized `ProbeSet` is

$$\sum_{i=1}^{p^*} L(f_i) + \sum_{X \in W \setminus V} S(|X[j_{X,0} :]|) \tag{3}$$

The first summation term is the total time spent in reading posting lists of the prefix, where p^* is the final prefix length and $p^* = |Q| - |Q \cap X_k^*| + 1$, where X_k^* is the final k-th candidate in the final result. The second summation term is the total time spent in reading all qualified candidates ($W \setminus V$) encountered in the prefix, where W is the set of all candidates, and V is the set of candidates pruned by the position filter. The position $j_{X,0}$ is the first matching token position of X with Q. Lastly, $S(\cdot)$ is the time of reading a set as a function of its size.

3 A NEW FRAMEWORK

`ProbeSet` takes advantage of the prefix filter to read fewer posting lists, but requires extra work, which is partially reduced by using position filter, to verify the candidates by reading and computing exact intersection sizes. In this section, we present a novel framework that quantifies the benefits (i.e., reduced cost) of reading posting lists and candidates, and an algorithm, called `JOSIE`, that uses this framework.

3.1 To Read or Not to Read

Example 4. *Let us revisit Ex. 3. Suppose after reading posting list x_2: instead of reading X_2 before X_3 and X_4, let us read X_4 first. The running heap after reading X_4 is $\{(X_1, 3), (X_4, 2)\}$, and by using position filter, $|Q \cap X_2|_{ub} = 1 + \min(4-2, 2-1) = 2 \leq 2$, and $|Q \cap X_3|_{ub} = 1 + \min(4-2, 1-1) = 1 < 2$, we can skip both X_2 and X_3 before terminating.*

In this example, we read only 2 sets X_1 and X_4, compared to three sets in Example 3. The take away is that we do not have to read a candidate immediately after we counter it. By prioritizing reading some candidates before the others, we can "lift" the running k-th intersection size higher, increasing the pruning power of the position filter, and read fewer candidates.

Example 5. *Now suppose we make a different change after reading posting list x_2: instead of reading any sets, we continue to read posting list x_{100}.*

The posting list x_{100} does not give us any new candidates – both X_1 and X_4 are seen, however, the new position information allows us to update the position filters for the seen candidates: for X_4, its last token we saw before x_{100} was x_2, thus no intersecting tokens between x_2 (position 1) and x_{100} (position 99), and the only possible intersecting tokens exist in the remaining $100 - 99 = 1$ token after x_{100}. Using this new information, the position filter of X_4 becomes $|Q \cap X_4|_{ub} = 1 + 1 + \min(4-3, 100-99) = 3$, and most importantly, we only need to read the last token from X_4 (not the whole set) to compute its exact intersection size. The rest is the same as Example 4: we can safely ignore X_2 and X_3 using position filters, and terminate. The complete read sequence is shown in Figure 1.

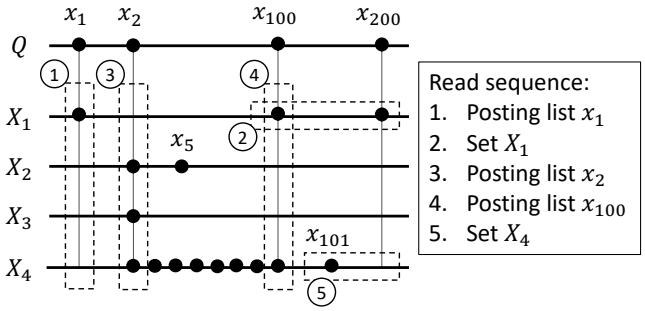

Figure 1: An example read sequence that alternates between reading posting lists and candidates. Each set is a horizontal line, and each posting list is a vertical line, which connects common tokens among sets.

In this example, we read in total three tokens from candidates, as shown in Figure 1: 2 from X_1 and 1 from X_4, while in Example 3 we read $2 + 1 + 99 = 102$ tokens and in Example 4 we read $2 + 99 = 101$ tokens. This example brings up two points: first, we do not need to read all candidates before reading the next posting list, as long as we come back to them (when necessary) before terminating search, we are still guaranteed to find the exact top-k; second, as Figure 1 shows, reading a posting list (e.g., x_{100}) has the benefit of potentially improving the position filters of unread candidates, in addition to reducing the amount of data (i.e., number of tokens) we have to read from candidates to compute exact intersection sizes.

As we can see from these two examples, reading a candidate and reading a posting list each has its own benefit in terms of reducing the time spent in reading other posting lists or candidates. These benefits have not been defined by the previous work and are data dependent. A quantitative comparison of these benefits leads to an approach that alternates between reading posting lists and reading sets, as illustrated by Figure 1. In the following sections, we will analyze these benefits quantitatively using a framework based on statistical approximation techniques, and present an adaptive algorithm for exact top-k overlap set similarity search that utilizes this framework.

3.2 Reading Candidates

As shown in Example 4, one potential benefit of reading a candidate is to increase the running k-th intersection size (i.e., increase $|Q \cap X_k|$) which is used by both the prefix filter and position filter. Recall, the prefix filter prunes out posting lists (those beyond the prefix) and all *unseen* candidates in these lists. On the other hand, the position filter prunes out *seen* candidates without reading them, by comparing the upper-bound intersection size of a candidate with threshold $|Q \cap X_k|$.

Now the problem is how do we know whether a candidate can make it into the running top-k and increase the threshold $|Q \cap X_k|$? Let X be the current candidate. Should we read it? We do not have full information about all the tokens in X, however, we actually know quite a lot:

- $i_{X,0}$ is the position in Q of the first token that intersects with X, where X first appears in the posting list of that token;
- $j_{X,0}$ is the position in X of the first token that intersects with the query;
- j_X is the position in X of the most recent token that intersects with the query; and
- $|Q[1:i] \cap X|$ is the number of intersecting tokens between Q and X up to and including the token at the current position i in Q, where $Q[1:i]$ is the prefix of Q up and including token x_i: $x_1, x_2, ..., x_i$.

Both j_X and $|X|$ can be found in the posting list entry of X. We read posting lists of Q's tokens from left to right according to the global token order, and keep track of $i_{X,0}$, j_X, $|Q[1:i] \cap X|$ and $|X|$ for all seen candidates. As explained in the previous section, we do not have to read every candidate as we encounter it, as long as we read it or prune it before terminating. Figure 2 illustrates.

Given this information about X, we can estimate its intersection $|Q \cap X|$, which, together with the running top-k heap, can be used to approximate the new threshold $|Q \cap X_k|$.

If we consider the set $Q \cap X$ as a subset of $Q[i_{X,0}:]$ – the subset of Q starting from the first matching token with X, and assume the members of $Q \cap X$ is uniformly distributed over $Q[i_{X,0}:]$, an unbiased estimator for $|Q \cap X|$ is

$$\begin{aligned} |Q \cap X|_{est} &= \frac{|Q[i_{X,0}:i] \cap X|}{i - i_{X,0} + 1} \times (|Q| - i_{X,0} + 1) \\ &= \frac{|Q[1:i] \cap X| - \cancel{|Q[1:i_{X,0}] \cap X|}}{i - i_{X,0} + 1} \times (|Q| - i_{X,0} + 1) \\ &= \frac{|Q[1:i] \cap X|}{i - i_{X,0} + 1} \times (|Q| - i_{X,0} + 1) \end{aligned} \tag{4}$$

if we consider $Q[i_{X,0}:i]$ as a random sample of $Q[i_{X,0}:]$. A proof of this can be done using techniques introduced in Broder [4]. If the sample $Q[i_{X,0}:i]$ is small, the variance will be high. Therefore, we should read a few posting lists starting from position $i_{X,0}$ before using this estimation.

Armed with the knowledge of $|Q \cap X|_{est}$, we can estimate the new threshold $|Q \cap X'_k|$, where X'_k is the new k-th candidate after reading X. We first compare $|Q \cap X|_{est}$ with the current threshold $|Q \cap X_k|$. If $|Q \cap X|_{est} \leq |Q \cap X_k|$, then we assume the candidate X likely will not qualify for the running top-k, and the current threshold is unchanged. If $|Q \cap X|_{est} > |Q \cap X_k|$, we then look at what the new threshold would be after pushing X to the heap (and popping X_k), by comparing $|Q \cap X|_{est}$ with $|Q \cap X_{k-1}|$, where X_{k-1} is the $(k-1)$-th candidate (done in constant time with a binary heap). The estimation can be summarized using Equation 5 below.

$$|Q \cap X'_k|_{est} = \begin{cases} |Q \cap X_k| & \text{if } |Q \cap X|_{est} \leq |Q \cap X_k| \\ |Q \cap X|_{est} & \text{if } |Q \cap X|_{est} > |Q \cap X_k| \\ & \text{and} |Q \cap X|_{est} \leq |Q \cap X_{k-1}| \\ |Q \cap X_{k-1}| & \text{if } |Q \cap X|_{est} > |Q \cap X_{k-1}| \end{cases} \tag{5}$$

Now using the new threshold, we can finally estimate the benefit of reading candidate X in terms of time saved. As mentioned earlier, the benefit consists of two parts. The first part is from eliminated posting lists through the prefix filter update. Let the current prefix length be $p = |Q| - |Q \cap X_k| + 1$, and the new prefix length after reading X be $p' = |Q| - |Q \cap X'_k|_{est} + 1$. Then the posting lists from $p' + 1$ to p inclusive will be eliminated, and the benefit is equal to the sum of the time to read these posting lists.

Another benefit is from eliminating candidates by updating the position filter. The position filter for a candidate Y has upper-bound intersection size:

$$|Q \cap Y|_{ub} = |Q[1:i] \cap Y| + \min(|Q| - i, |Y| - j_Y) \tag{6}$$

This is different from Equation (2): the first term on the right is the number of intersecting tokens seen so far, as we do not read Y immediately after encountering it, this number may no longer be one. If $|Q \cap Y| \leq |Q \cap X'_k|_{est}$, then candidate Y will likely be eliminated after reading X. Thus, using the updated position filter, we can determine whether a candidate will be eliminated, and the benefit is the sum of the time it would take to read the eliminated candidates.

Equation 7 gives the benefit of reading candidate X. The set W_i is all unread candidates at posting list i and $I(\cdot)$ is an indicator function that evaluates to one only if the condition argument is true, and zero otherwise.

$$\begin{aligned} Benefit(X) = &\sum_{i=p'+1}^{p} L(f_i) + \\ &\sum_{Y \in W_i, Y \neq X} S(|Y[j_Y + 1:]|) \cdot I(|Q \cap Y|_{ub} \leq |Q \cap X'_k|_{est}) \end{aligned} \tag{7}$$

3.3 Reading Posting Lists

We now provide a quantitative framework for estimating the benefit of reading posting lists.

As before, let i be the position of the current posting list that we just read. As discussed in the previous section, in order to avoid a large variance in estimating intersection size, we need to initially read a few posting lists for each candidate. So posting lists are read in "batch", and we use i' to indicate the position of the end, or the last posting list, of the

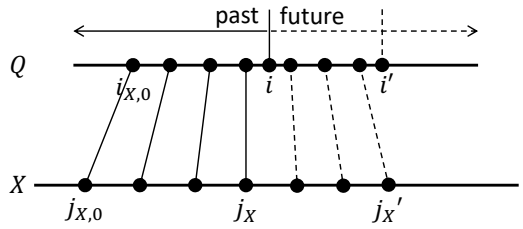

Figure 2: The query set Q and candidate X after reading the posting list at position i. Pairs of dots with solid lines represent intersecting tokens, pairs with dotted lines represent future intersecting tokens, and non-intersecting tokens in both sets are not shown.

next batch. This is shown in Figure 2, which also shows the intersecting tokens between query Q and a candidate X. Let j_X be the most recent position in X whose token intersects with Q, and j'_X be the position in the future after reading the next batch of posting lists ending at i'.

Reading the next batch always improves (or at worse leaves unchanged) the pruning power of position filter, by tightening upper-bound for all candidates. This can be understood using the future upper-bound intersection size for candidate X at i':

$$\begin{aligned}
|Q \cap X|'_{ub} &= |Q[1:i'] \cap X| + \min(|Q| - i', |X| - j'_X) \\
&= |Q[1:i] \cap X| + |Q[i+1:i'] \cap X| \\
&\quad + \min(|Q| - i', |X| - j'_X) \\
&\leq |Q[1:i] \cap X| + |Q[i+1:i']| \\
&\quad + \min(|Q| - i', |X| - j'_X) \\
&\leq |Q[1:i] \cap X| + \min(|Q| - i, |X| - j_X) \\
&= |Q \cap X|_{ub}
\end{aligned} \tag{8}$$

Intuitively, reading the next batch verifies the exact intersection in that batch, and "exposes" the number of tokens that do not intersect but have taken as part of the upper-bound. So the total number of tokens we can count toward the upper-bound is reduced, and the upper-bound is lowered.

Given the new upper-bound $|Q \cap X|'_{ub}$, by comparing it with the current k-th candidate's intersection size, or the current threshold, $|Q \cap X_k|$, we know for every candidate X whether it would be eliminated or not. If not, we may still have some benefit because we know we would only need to read the tokens in $X[j'_X + 1:]$.

So now the question is how do we estimate $|Q \cap X|'_{ub}$. Calculating $|Q \cap X|'_{ub}$ requires $|Q[i+1:i'] \cap X|$, the number of intersecting tokens between Q and X in the next batch, and j'_X, the future position of the last intersecting token.

First, we can estimate $|Q[i+1:i'] \cap X|$ using the same method we used for the total intersection size, $|Q \cap X|$, in Equation 4, by assuming uniform distribution of intersecting tokens over the range $Q[i_{X,0}:]$.

$$|Q[i+1:i'] \cap X|_{est} = \frac{|Q[i_{X,0}:i] \cap X|}{|Q| - i_{X,0} + 1} \cdot (i' - i) \tag{9}$$

Since we keep track of $i_{X,0}$ and $|Q[i_{X,0}:i] \cap X|$ for every candidate X, we can compute this estimate.

Second, we can estimate j'_X by first leveraging the fact that the number of intersecting tokens between $i+1$ and i' in Q must be equal to that between $j_X + 1$ and j'_X in X, as shown in Figure 2. Then, we can apply the same estimation method for intersection size in Equation 9 to create an approximate equality, from which we derive an expression for $j'_{X,est}$.

$$\begin{aligned}
|Q[i+1:i'] \cap X|_{est} &\approx |Q \cap X[j_X + 1:j'_X]|_{est} \\
\frac{\cancel{|Q[i_{X,0}:i] \cap X|}}{|Q| - i_{X,0} + 1} \cdot (i' - i) &\approx \frac{\cancel{|Q \cap X[j_{X,0}:j_X]|}}{|X| - j_{X,0} + 1} \cdot (j'_X - j_X) \\
j'_{X,est} &= j_X + \frac{i' - i}{|Q| - i_{X,0} + 1} \cdot (|X| - j_{X,0} + 1)
\end{aligned} \tag{10}$$

In the above derivation, since $|Q[i_{X,0}:i] \cap X|$ is exactly the same as $|Q \cap X[j_{X,0}:j_X]|$ (see Figure 2), we can cancel them out on both sides of the equation.

Substituting $|Q[i+1:i'] \cap X|_{est}$ and $j'_{X,est}$ for $|Q[i+1:i'] \cap X|$ and j'_X, respectively, in Equation 8, we can obtain the estimation for the future upper-bound of X in the position filter:

$$\begin{aligned}
|Q \cap X|'_{ub,est} &= |Q[1:i] \cap X| + |Q[i+1:i'] \cap X|_{est} \\
&\quad + \min(|Q| - i', |X| - j'_{X,est})
\end{aligned} \tag{11}$$

Equipped with the estimation for the upper-bound, we can finally summarize the benefit of reading the next batch of posting lists for tokens $B_{i+1,i'} = x_{i+1}, ..., x_{i'}$.

$$\begin{aligned}
&Benefit(B_{i+1,i'}) \\
&= \sum_{X \in W_i} S(|X[j_X:]|) \cdot I(|Q \cap X|'_{ub,est} \leq |Q \cap X_k|) \\
&+ \big(S(|X[j_X:]|) - S(|X[j'_X:]|)\big) \cdot I(|Q \cap X|'_{ub,est} > |Q \cap X_k|)
\end{aligned} \tag{12}$$

The first term is the time spent on reading candidate X, corresponding to the cost saved due to the elimination of X by reading the next batch of posting lists. The second term is the reduction in read time for reading X when it is not eliminated, but due to an updated position filter, fewer of its tokens need to be read. As in Equation 7, $I(\cdot)$ is an indicator function. It is important to note that, the benefit of reading posting lists is always non-negative.

3.4 An Adaptive Algorithm

Given our quantitative framework for estimating the respective benefits of reading a candidate and reading a batch of posting lists (in terms of the amount of read time saved), we

Algorithm 1 Algorithm JOSIE using the cost model

```
1: procedure JOSIE(U, I, Q, k, λ) ▷ U, I are the dictionary and inverted index, and λ is the batch size
2:     Q ← Q ∩ U                                   ▷ Use the dictionary
3:     x_1, x_2, ..., x_n ← SORT(Q)                ▷ Apply global ordering
4:     h ← {}                                      ▷ h is the heap for running top-k sets
5:     u ← {}                                      ▷ u is the hash map of unread candidates
6:     i ← 1, i' ← 1 + λ, X_k ← ∅ ▷ X_k is the k-th candidate
7:     p ← |Q| − |Q ∩ X_k| + 1
8:     while i ≤ p or u ≠ ∅ do
9:         X ← BEST(u)
10:        if |h| = k and NETCOST(X) > NETCOST(B_{i+1,i'}) or (u = ∅ and X = ∅) then
11:            u ← u ∪ READ(I, B_{i+1,i'})          ▷ Read posting lists
12:            i ← i', i' ← i' + λ
13:        else
14:            TRYPOPPUSH(h, k, X, |Q ∩ X|)         ▷ Read set X
15:            X_k, |Q ∩ X_k| ← HEAD(h)
16:            p ← |Q| − |Q ∩ X_k| + 1
17:        end if
18:        u ← POSITIONFILTER(u)                    ▷ Eliminate candidates
19:    end while
20:    return POPALL(h)
21: end procedure
```

now calculate the *net cost* as the difference between the read time incurred and the read time saved.

$$\begin{aligned} NetCost(X) &= S(|X[j_X + 1 :]|) - Benefit(X) \\ NetCost(B_{i+1,i'}) &= \sum_{x=i+1}^{i'} L(f_x) - Benefit(B_{i+1,i'}) \end{aligned} \tag{13}$$

Clearly, the lower the net cost, the better the performance.

We now present our algorithm, JOSIE (**JO**ining **S**earch using **I**ntersection **E**stimation), that prioritizes reading posting lists or candidates based on net cost computed using estimated intersection sizes. The pseudo code is shown in Algorithm 1.

Starting from reading the first batch of posting lists, the algorithm relies on the functions NETCOST(X) and NETCOST($B_{i+1,i'}$) to determine whether to read the best unread candidate, ranked by net cost, or to read the next batch of posting lists. Each time candidate is read, the algorithm updates a running top-k heap and the prefix filter. Each time posting lists are read, the algorithm updates the pointers. Either way, after the read, the algorithm applies the position filter and eliminates unqualified candidates.

Since the threshold $|Q \cap X_k|$ is only valid (or non-zero) after at least k candidates are read, JOSIE always chooses to read candidates before the top-k heap is full ($|h| = k$), unless it is reading the first batch ($u = \emptyset$ and $X = \emptyset$).

The algorithm terminates when the posting list pointer i is outside of the prefix filter range ($i > p$), and there is no unread candidate ($u = \emptyset$). If $i > p$ but $u \neq \emptyset$, it means that the algorithm still has to finish the remaining candidates by either reading posting lists or reading sets.

The total time of JOSIE is

$$\sum_{i=1}^{p^*} L(f_i) + \sum_{i=p^*+1}^{p^*+\delta} L(f_i) + \sum_{X \in W \setminus V^*} S(|X[j_X :]|) \tag{14}$$

The formula is similar to ProbeSet's (Equation 3) however it has a few differences. First, in ProbeSet, no more posting lists are read after the last position in the final prefix, indicated by p^*, while in our algorithm, the net cost of reading the next batch of posting lists may be less than reading the next candidate, leading to more posting lists read. We use δ to indicate the extra posting lists read after the final prefix position. It is important to note that the extra posting lists are only used to help reduce the read time of existing candidates, and do not introduce any new candidate, as those candidates will be pruned automatically by the prefix and position filters.

The second difference is the last term. Instead of reading every qualified candidate as we encounter it as in ProbeSet, we read candidates in increasing order of their net costs, and thus the candidates with the highest pruning effect (i.e., benefit) get read first. Thus, we are always going to read no more candidates than ProbeSet and often many fewer. That is, $V \subseteq V^*$, where V is the set of candidates that are pruned at first encounter only, and V^* is the set of all candidates that are pruned in our algorithm. Note that $V^* = \emptyset$ is extremely unlikely as it happens only when the candidates appear in the order of strictly increasing intersection sizes with the query.

The last difference is the read time for each candidate. ProbeSet reads a qualified candidate at first encounter, thus the read time is $S(|X[j_{X,0} :]|)$. Our algorithm, on the other hand, reads a candidate at some posting list after the first encounter, due to reading in batch and using the cost model. Thus, we will almost always have fewer tokens to read for any candidate (in the worst case read the full set like ProbeSet). Thus less time is spent on reading the candidates.

In conclusion, our algorithm reads fewer candidates than ProbeSet, and often a smaller portion of those candidates. This reduction in read time is at the expense of reading the extra δ number of posting lists. However, due to the use of cost model, every step of the algorithm chooses the direction with the least net cost, thus the total cost of reading the extra posting lists is likely much less than the reduced cost of reading candidates.

3.5 Distinct Posting Lists

Although the cost model works with any global order, for this work, we use increasing frequency order as the global order, because it tends to minimize the number of candidates in the prefix, as suggested by Chaudhuri et al. [6]. For details on indexing, please see Appendix 7.1. Another advantage of using frequency order is that duplicate posting lists are together, and we use this to further optimize the search engine.

Two posting lists are duplicates if they point to the same sets. Because of the frequency ordering (and within a frequency we order lists by some fixed ordering on sets), duplicate lists will be adjacent. Importantly, we observed many duplicate posting lists in both Open Data and Web Table. This is due to many data values appear only once in a single column (e.g., UUIDs), or they strictly co-occur with some other values, such as provinces' names of a country. The statistics are in Table 3.

Table 3: Posting lists in Open Data and Web Tables.

	#Original	%Dup	#AfterDe-dup
Open Data	563,320,456	98%	9,003,658
Web Tables	184,644,583	83%	45,395,793
Enterprise Data	3,902,604	99.87%	9,133

Even though reading multiple duplicate posting lists does not yield more candidates than reading just one, it still provides the benefit of reducing the cost of reading existing candidates, as discussed in Section 3.3. So how can we avoid reading duplicate lists, while still getting the benefit?

To avoid reading duplicate posting lists, we assign each token in the dictionary a *duplicate group ID*, which is a unique identifier for a group of duplicate posting lists. So when matching a query with the dictionary, a duplicate group ID is also mapped to every query token, in addition to the frequency and posting list pointer.

The cost model assumes the posting lists of a query set are read sequentially in the frequency order of their tokens. When going through the posting lists, we read just the posting list of the last token in each group present in the query. This can be done by checking if the next token has the same or different duplicate group ID as the current one. If the next one has the same duplicate group ID, then the posting list can be skipped. This idea can be illustrated using the example in Figure 3: the first and second posting lists with duplicate group ID 1 are skipped, and the first two posting lists with duplicate group ID 5 are also skipped.

We modified our cost model to handle duplicates. First, we count groups, rather than lists when forming batches and ignore the skipped posting list in calculating the read cost. Second, we need to account for the number of posting lists skipped when reading the last posting list in the same duplicate group. So the information about candidates (see Section 3.2), such as the number of intersecting tokens observed, is updated correctly.

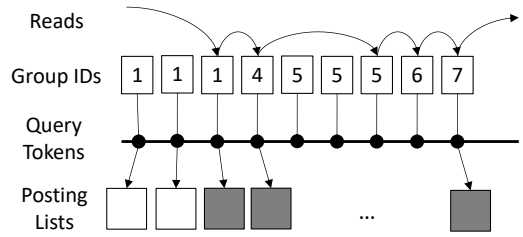

Figure 3: A read sequence of distinct posting lists.

4 EXPERIMENTS

We now demonstrate the performance of JOSIE through experiments using real-world data lakes, and compare it to the state-of-the-arts in both exact and approximate approaches.

4.1 Data Lakes

We used two data lakes in our experiments, Open Data (obtained from Nargesian et al. [21]) and WebTables, (a public corpus [14]).

For each lake, we extracted sets by taking the distinct values in every column of every table. We also removed all numerical values, as they create casual joins that are not meaningful. The characteristics of the extracted sets are shown in Table 2. Web Tables has 219× more sets than Open Data, while its average set size is much smaller (10 vs. 1,540). This means reading a set in Web Table is usually cheaper. Tokens in Web Tables appear in more sets overall than those in Open Data. This implies that the posting lists of Web Tables tokens are often longer, and thus more expensive to read.

4.2 Query Benchmarks

We generated three query benchmarks from each data lake. Each benchmark is a set of queries (sets) selected from a size range in order to evaluate performance on different query sizes.

For Open Data, the benchmarks are from three ranges: 10 to 1k, 10 to 10k, and 10 to 100k. For the rest of the paper, we will refer to each benchmark using its upper-bound. For example, the benchmark with range 10 to 100k is simply called "100k". For each benchmark, we divide its range into 10 equal-width intervals except for the first interval, which starts from 10. For example, for range 10 to 1k, the intervals are $[10, 100]$, $[100, 200]$, and all the way to $[900, 1000]$. We

then sample 100 sets from each interval using uniform random sampling. Sampling by interval prevents a benchmark that has the same skewed distribution as the repository itself, and heavily biased to sample smaller sets.

For Web Tables, we used different ranges for benchmarks: 10 to 100, 10 to 1k, and 10 to 5k. The last range only has 5 intervals. This is because the sets in Web Tables tend to be much smaller than Open Data, and there are not enough sets in the 5k to 10k range to sample 100 sets for every interval. In order to make up for the total, we sampled 200 sets for each of the 5 intervals.

For a query set Q, and dictionary of all tokens U in the index excluding the query, we used $|Q \cap U|$ instead of $|Q|$ to decide which range the query belongs to. This is because any token that only exists in Q, but not in U will not have a posting list, and add uncontrolled noise to the experiment when we want to measure the effect of number of posting lists on performance.

We have made the benchmarks available online[5].

4.3 Setup

We build an inverted index and dictionaries for each of the Open Data and Web Tables repositories using Apache Spark[6]. The details of index creation is discussed in Appendix 7.1. The posting lists and sets are then stored in a PostgreSQL[7] database as two separate tables, with BTree indexes built on tokens and sets. All experiments are conducted on a machine with two Intel® Xeon® CPU E5-2620 v4 @ 2.10GHz (16 cores), 128 GB DDR4 memory, and an Intel® SSD DC S3520 3D MLC.

4.4 Algorithms

`MergeList-D` This is based on the `MergeList` algorithm introduced in Section 2.2, modified to read only the distinct posting lists using the technique introduced in Section 3.5.

`ProbeSet-D` This is based on the `ProbeSet` algorithm introduced in Section 2.4, modified to also read only distinct posting lists. This algorithm relies on prefix filter to limit the number of posting lists to read, and uses position filter to reduce the number of candidates to read.

`LSHEnsemble-60` This is an implementation of the algorithm introduced by Zhu et al. for approximate joinable table search [34]. It uses a transformation that allows Minhash LSH to be used with set containment similarity, $\frac{|Q \cap X|}{|Q|} \rightarrow R[0.0, 1.0]$, which is the normalized version of set intersection size. The algorithm first uses an LSH index to acquire some candidates that are more likely than the rest to have set containment similarities above some threshold, and then the algorithm reads and computes exact set intersection size for those candidates to obtain the ranked final results. The original algorithm[8] only supports threshold-based search, so we modified the algorithm to support top-k search by trying a sequence of decreasing thresholds starting at 1.0 with a step size equal to 0.05, until enough candidates have been acquired. Since this is an approximate algorithm, it can be extremely fast by just returning any candidate. So for a fair comparison we require the algorithm to keep acquiring and reading (distinct) candidates until a minimum recall of 60% is reached for the top-k candidates. The ground truth is provided to the algorithm for every query.

`LSHEnsemble-90` This is the same algorithm as the previous one, with the only difference being the minimum recall is 90%. A higher minimum recall means the algorithm has to read more candidates in order to improve accuracy.

`JOSIE-D` This is the cost-based algorithm presented in Section 3.4, with the distinct posting list optimization.

Figure 4: Mean query time, Open Data benchmark.

4.5 Open Data Results

We compare the running time of `JOSIE-D` with the baseline algorithms `MergeList-D` and `ProbeSet-D` on benchmarks generated from Open Data tables. The result is shown in Figure 4. Each row corresponds to a different k, which is the number of top results to retrieve, and each column corresponds to a different query benchmark (i.e., 1k, 10k, and 100k). Each point is the mean running time of benchmark queries from a single intervals (i.e., 100 queries).

[5]https://github.com/ekzhu/set-similarity-search-benchmarks
[6]https://spark.apache.org
[7]https://www.postgresql.org
[8]https://github.com/ekzhu/lshensemble

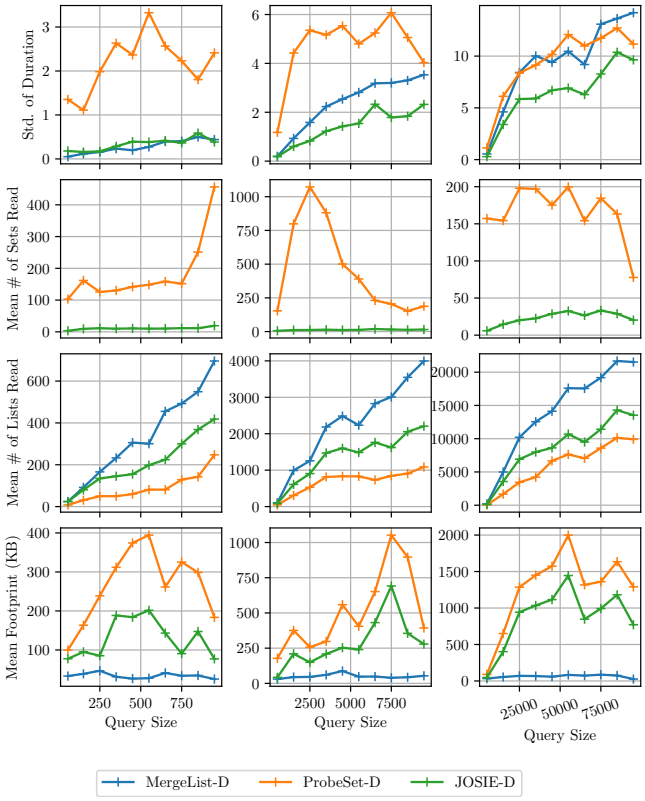

Figure 5: Standard deviation of durations, mean number of set and posting lists read per query, and query memory footprint on Open Data benchmark ($k = 10$).

ProbeSet-D's performance deteriorates quickly as k increases. This is because with larger k, the prefix filter is larger and has less pruning power, as the k-th candidate's intersection size can be much lower. Thus, ProbeSet-D must read more posting lists and all the candidates appear in those posting lists. In contrast, MergeList-D simply reads all the distinct posting lists and none of the candidates, regardless of k. So its performance is constant with respect to k.

We can also observe that ProbeSet-D's running time is much more volatile than MergeList-D, as shown in the first row of Figure 5, which plots the standard deviation of query duration. This is because ProbeSet-D reads all candidates it encounters and has no control over the sizes of candidates it reads, thus its running time is heavily dependent on the distribution of posting list lengths (i.e., token frequencies) and set sizes.

Now we look at our proposed algorithm JOSIE-D. It outperforms both ProbeSet-D and MergeList-D, on all benchmarks and all ks, often by 2 to 4 times, except for the $k = 20$ and 1k benchmark, on which it is on par with MergeList-D.

Let us discuss the effect of increasing k on JOSIE-D, as it also reads candidates, similar to ProbeSet-D. The running time of JOSIE-D increases with k, but at a much slower rate than ProbeSet-D. This is because unlike the latter, JOSIE-D does not read all the candidates it encounters – it always reads the next candidate with the lowest net cost (and likely carrying the most pruning power, or benefit) evaluated by the cost model, and aggressively prunes out unqualified candidates using the position filter.

We compare the mean number of sets read by ProbeSet-D and JOSIE-D per query in the second row of Figure 5. It is evident that JOSIE-D drastically reduced the sets read by an order of magnitude (from well over 100 to around 30). This shows that the cost model is extremely effective in choosing the best next candidate to read, which causes the most aggressive pruning. We also compare the mean number of posting lists read in the third row of Figure 5, which shows that JOSIE-D reads more posting lists than ProbeSet-D. This is because the net cost of reading posting lists may be lower than reading the next best candidates, as discussed in Section 3.3.

What happened at $k = 20$ on the 1k benchmark? The estimation time becomes a major fraction of the total time when read time is small – this happens when the candidates are small in size, but large in total number as happens with small queries at large k. The estimation time is linear in the total number of candidates, as JOSIE-D must always go through all unread candidates. In this case, MergeList-D may benefit from the fact that it does not read any candidate, and the number of posting lists to read is also small for small queries.

We also compares the memory footprint of query processing for the three algorithms, in the fourth row of Figure 5. For the detail of memory footprint measurement, please refer to Appendix 7.3. Based on the results, ProbeSet-D uses the most memory, and JOSIE-D comes the second. This is due to the large sets in the Open Data benchmark, as the two algorithms both need to allocate buffer for reading candidate sets. The relatively short posting lists in the Open Data benchmark also lead to the low memory usage of MergeList-D.

4.6 Webtable Results

In comparison with Open Data, Web Tables has different characteristics – it has 219× more sets, but much smaller sets (average size 10 versus 1,540), and its tokens have higher frequencies overall (average 4 versus 23), leading to larger posting lists that are more expensive to read.

The effect of large posting lists directly impacts the performance of MergeList-D, which must read all distinct posting lists for a query set. As shown in Figure 6, it has the longest running time, and can be 3× slower than ProbeSet-D and

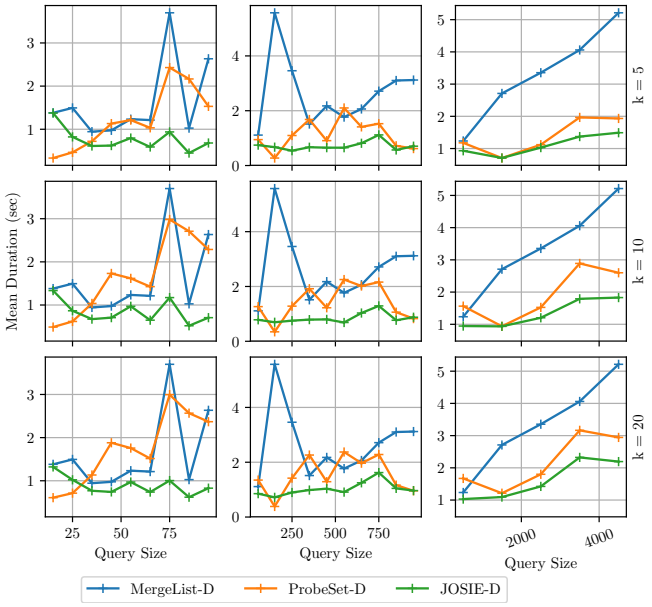

Figure 6: Mean query time on WebTable benchmark.

JOSIE-D on $k = 5$. On the other hand, ProbeSet-D benefits from the small set sizes in Web Tables, as the time saved from reading fewer large posting lists is larger than the time to read tiny sets. So in consequence, for table repositories with similar distributions to Web Table – high token frequencies and small set sizes, search algorithms making use of prefix filter (e.g., ProbeSet-D and JOSIE-D) would benefit the most from the distribution.

What about JOSIE-D? It still out-performs the other two algorithms, as shown in Figure 6, with lower variance as shown in the first row of Figure 7. The most interesting part is that it still mostly out-performs ProbeSet-D despite the high estimation time caused by having many candidates to look through. The second row of Figure 7 shows that the number of candidates in Web Tables is much higher than Open Data, approximately 15× on the 1k benchmark, for example. JOSIE-D still reads an order of magnitude fewer candidates than ProbeSet-D, so the estimation time is mostly paid-off by the time saved from pruned candidates.

The fourth row of Figure 7 shows the memory footprint of the three algorithms on Web Table benchmark. The result is very different from the Open Data benchmark: MergeList-D comes first in memory usage, exceeding the other two by an order of magnitude for query size larger than 250. This is due to the relatively much longer posting lists in the Web Table benchmark, making it much more costly space-wise to read posting lists. The smaller set sizes reduce the memory usage of both ProbeSet-D and JOSIE-D, however, because the latter still reads more posting lists (δ more, as explained in Section 3.4), it tends to use more memory.

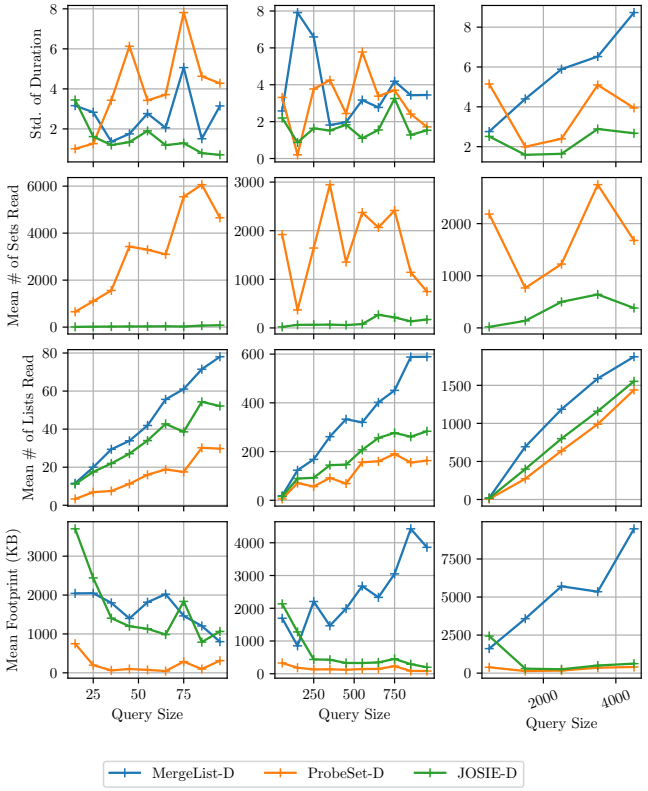

Figure 7: The standard deviation, number of sets and posting lists read, and query memory footprint on Web Table benchmark ($k = 10$).

One interesting result is that JOSIE-D uses a lot more memory for smaller queries than larger ones. This is because the algorithm tends to read posting lists rather than sets for small queries, as shown in the second row of Figure 7, and posting lists requires larger buffer than sets in this benchmark. JOSIE-D also uses more memory than MergeList-D for those small queries, because unlike MergeList-D, it needs to read token positions and set sizes in addition to set IDs (see Appendix 7.3).

4.7 Comparison with LSH Ensemble

The most interesting comparison comes from JOSIE-D and LSH Ensemble algorithm (LSHEnsemble-60 and 90). Figure 8 shows the mean query durations. The LSH Ensemble query duration includes the time of retrieving candidates from the LSH index and the time to read and compute exact intersection sizes for candidates. The common key to the success of LSH algorithms is that the first stage of candidate retrieval is extremely fast and can be very selective, such that the total query duration can be lower than exact algorithms.

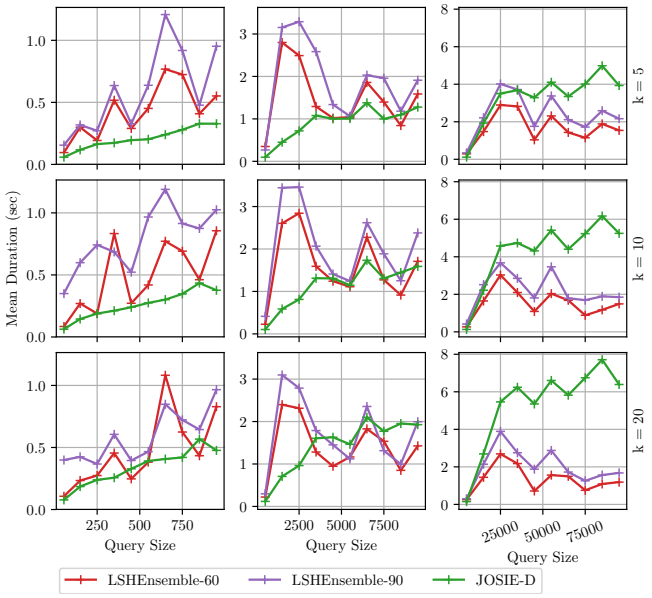

Figure 8: Mean query time of LSH Ensemble and JOSIE on Open Data benchmark.

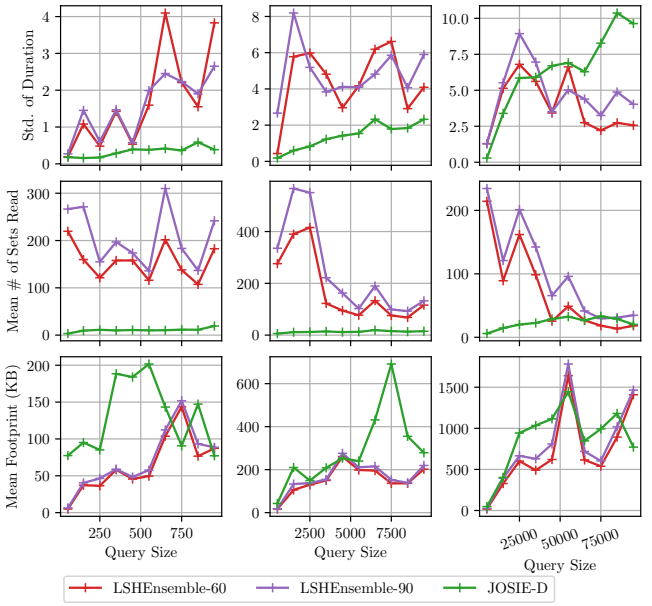

Figure 9: The standard deviation, number of sets read and query memory footprint on Open Data benchmark for $k = 10$.

However, based on Figure 8, we observe the performance of JOSIE-D can beat LSH Ensemble by 2× on the 1k benchmark for $k = 5$ to $k = 10$. This is surprising, because LSH Ensemble is an approximate algorithm (though we have modified it to achieve a minimum 60% recall for LSHEnsemble-60 and 90% for -90) while JOSIE-D gives exact results. As query size increases, for example, on the 100k benchmark, LSH Ensemble begins to out-perform JOSIE-D by a large margin. This is also true with increasing k, as at $k = 20$ on the 100k benchmark, LSH Ensemble out-performs JOSIE-D by 3.5×.

The reason for JOSIE-D being faster at smaller k is actually related to observations made by Bayardo et al. who showed experimentally that prefix filter based exact techniques can be faster than LSH [2]. This is because prefix filter can be much more selective than LSH index at high thresholds or small k, leading to very few posting list reads and smaller number of candidates reads.

Why is LSH Ensemble slower than JOSIE-D for small queries? Our investigation shows that this is because the transformation from containment similarity (i.e., normalized intersection size $\frac{|Q \cap X|}{|Q|}$) to Jaccard similarity used by LSH Ensemble introduces additional false positive candidates [34], which causes wasted time spent in reading those sets. We found that this influx of false positive candidates is the most severe at small queries, due to the skewed set size distribution where there are many more small sets than large sets. The sudden drop of candidates can be observed in the second row of Figure 9, the plot for the 10k benchmark. Unlike JOSIE-D, LSH Ensemble cannot use position filter to prune out false positive candidates, it must read all candidates it encounters.

As shown in the 10k and 100k benchmarks of Figure 9, the number of candidates read by LSH Ensemble drops closer to that of JOSIE-D, while its running time becomes relatively faster than the latter. This is because LSH Ensemble does not need to read any posting lists, and the candidate retrieval time is purely in-memory and extremely fast, while JOSIE-D must issue read operations for posting lists to retrieve candidates.

The third row of Figure 9 compares the query memory footprint of LSH Ensemble and JOSIE-D. Because JOSIE-D needs to read posting lists in addition to sets, it is expected to use more memory than LSH Ensemble.

In conclusion, LSH Ensemble can be a better choice if the user wants to retrieve many results ($k \geq 20$) and the query size is large (more than 10k) and most importantly, if the user does not care very much about recall of the results. Otherwise, a cost-based exact algorithm such as JOSIE-D is the better choice.

5 RELATED WORK

The overlap set similarity search problem is a type of set similarity search where the similarity measure is set intersection size. A very similar problem is *set similarity join*, which is also know as all-pair set similarity search. Given a collection of sets, set similarity join finds all the set pairs whose similarity

is less than a threshold. There are many work on in-memory set similarity join. Mann et al. [18] present an excellent experimental study. Specifically, Arasu et al. [1] designed a filtering condition with guaranteed false positive rate while not missing any result. Bayardo et al. [2] proposed to use prefix filter for set similarity join. Xiao et al. [32] extended the work by Bayardo et al. [2] with two additional filters: the position filter, which we have discussed extensively in Section 2.4, and the suffix filter. We do not use the suffix filter given the later Mann et al. study showed it was not very effective (something we confirmed in our initial development of JOSIE). Instead of using a fixed-length prefix as done by Bayardo et al., Wang et al. [29] designed an adaptive prefix filter framework. Wang et al. [30] further improved Xiao et al.'s work by leveraging the relation between sets.

A lot of research on parallel set similarity join has been done [10, 20, 22, 24, 26, 27]. Fier et al. [12] present a recent experimental study of this work. Vernica et al. [27] designed a parallel algorithm based on the prefix filter. MassJoin [10] uses partition-based string similarity join [17]. ClusterJoin [24] partitions the sets and distributes the partitions to different nodes. The goal of these parallel approaches is to scale in the number of sets, not necessarily in the size of the sets. For example, in the comparative study of Fier et al., the maximum dictionary size is 8M.

In addition to threshold-based set similarity join algorithms, Xiao et al. [31] proposed an in-memory algorithm for top-k set similarity join with Jaccard/Cosine similarity measures. We have introduced the search version of this algorithm in Section 2.3. Deng et al. [9] and Wang et al. [28] studied the top-k string similarity search problem. Behm et al. [3] studied the string similarity search problem in external memory. Note that all these techniques are not designed for data lakes. They are typically evaluated on sets with small average size and small dictionaries.

There is also research on finding related (not necessarily joinable) tables in data lakes. Sarma et al. [23] use keyword search and similarity measures other than set intersection size to find tables that are candidates for join and candidates for union from Web Tables [5]. The Mannheim Search Engine applies keyword search techniques to find joinable attributes by treating attribute values as keywords and ranking candidate attributes using a fuzzy Jaccard similarity [15]. This work is not intended to scale to large sets. Lehmberg et al. [13] found that stitching small web tables can help match them with knowledge bases. Nargesian et al. [21] proposed techniques to search for unionable tables from data lakes. Deng et al. [8] designed a set relatedness metric which can be used to find related tables.

Approximate set similarity search techniques have been used for finding related tables. Both Asymmetric Minwise Hashing [25] and LSH Ensemble [34] use MinHash-based indexing for set containment search. LSH Ensemble was shown to handle data sets with skewed cardinality distribution much better than Asymmetric Minwise Hashing, hence we have used LSH Ensemble in our experimental study in Section 4. Fernandez et al. [11] use MinHash LSH to find similar tables based on Jaccard similarity between columns. For the skewed distributions we consider, Jaccard is not an appropriate measure.

6 CONCLUSION

We have presented JOSIE, a new exact overlap set similarity search algorithm. Unlike previous approaches, JOSIE uses a cost model to adapt to the data distribution enabling significantly better performance over data lakes with large sets than the state-of-the-art approach ProbeSet. For queries with sizes up to 10K (a larger size than has been studied in the exact set similarity search literature), JOSIE even outperforms what is currently the best approximate approach. Our extensive experimental evaluation is the first to compare overlap set similarity search approaches over a repository with sets of average size over 1K and maximum size in the millions (two orders of magnitude larger than previous evaluations).

During the search, our approach estimates the likely set intersection size between a candidate and the query. Going forward, we are considering how to improve the estimation of set intersection size in a way that takes into account the frequency of tokens. We believe this would lead to better estimation of the net cost and make our cost model more accurate. In addition, we also looking into automate the selection of query column based on pre-computed statistics, and generalize the single-column model of equi-join to multi-column equi-joins. Lastly, fuzzy join is another interesting direction of future work.

Our work and our competitors for both exact and approximate search all use set intersection size to rank results. Of course, when searching relational tables, attributes may be bags. An open problem is to use bag semantics (including the multiplicities of tokens within a set) to rank results. This would allow a data scientist to be more informed about the relative usefulness of search results because it would reflect the actual size of the joined tables.

Acknowledgments: The work was partially funded by NSERC.

REFERENCES

[1] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. 2006. Efficient Exact Set-Similarity Joins. *PVLDB*, 918–929.

[2] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. 2007. Scaling up all pairs similarity search. In *WWW*. 131–140.

[3] Alexander Behm, Chen Li, and Michael J. Carey. 2011. Answering approximate string queries on large data sets using external memory. In *ICDE*. 888–899.

[4] A. Broder. 1997. On the Resemblance and Containment of Documents. In *SEQUENCES*. 21–.

[5] Michael J. Cafarella, Alon Y. Halevy, Daisy Zhe Wang, Eugene Wu, and Yang Zhang. 2008. WebTables: exploring the power of tables on the web. *PVLDB* 1, 1 (2008), 538–549.

[6] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. 2006. A Primitive Operator for Similarity Joins in Data Cleaning. In *ICDE*. 5.

[7] Dong Deng, Raul Castro Fernandez, Ziawasch Abedjan, Sibo Wang, Michael Stonebraker, Ahmed K. Elmagarmid, Ihab F. Ilyas, Samuel Madden, Mourad Ouzzani, and Nan Tang. 2017. The Data Civilizer System. In *CIDR*.

[8] Dong Deng, Albert Kim, Samuel Madden, and Michael Stonebraker. 2017. SilkMoth: An Efficient Method for Finding Related Sets with Maximum Matching Constraints. *PVLDB* 10, 10 (2017), 1082–1093.

[9] Dong Deng, Guoliang Li, Jianhua Feng, and Wen-Syan Li. 2013. Top-k string similarity search with edit-distance constraints. In *ICDE*. 925–936.

[10] Dong Deng, Guoliang Li, Shuang Hao, Jiannan Wang, and Jianhua Feng. 2014. MassJoin: A mapreduce-based method for scalable string similarity joins. In *ICDE*. 340–351.

[11] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, and Michael Stonebraker. 2018. Aurum: A Data Discovery System (To Appear). In *ICDE*.

[12] Fabian Fier, Nikolaus Augsten, Panagiotis Bouros, Ulf Leser, and Johann-Christoph Freytag. 2018. Set Similarity Joins on MapReduce: An Experimental Survey. *PVLDB* 11, 10 (2018), 1110–1122.

[13] Oliver Lehmberg and Christian Bizer. 2017. Stitching Web Tables for Improving Matching Quality. *PVLDB* 10, 11 (2017), 1502–1513.

[14] Oliver Lehmberg, Dominique Ritze, Robert Meusel, and Christian Bizer. 2016. A Large Public Corpus of Web Tables containing Time and Context Metadata. In *WWW*. 75–76.

[15] Oliver Lehmberg, Dominique Ritze, Petar Ristoski, Robert Meusel, Heiko Paulheim, and Christian Bizer. 2015. The Mannheim Search Join Engine. *Journal of Web Semantics* 35 (2015), 159–166.

[16] Chen Li, Jiaheng Lu, and Yiming Lu. 2008. Efficient Merging and Filtering Algorithms for Approximate String Searches. In *ICDE*. 257–266.

[17] Guoliang Li, Dong Deng, Jiannan Wang, and Jianhua Feng. 2011. PASS-JOIN: A Partition-based Method for Similarity Joins. *PVLDB* 5, 3 (2011), 253–264.

[18] Willi Mann, Nikolaus Augsten, and Panagiotis Bouros. 2016. An Empirical Evaluation of Set Similarity Join Techniques. *PVLDB* 9, 9 (2016), 636–647.

[19] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval*. Cambridge University Press.

[20] Ahmed Metwally and Christos Faloutsos. 2012. V-SMART-Join: A Scalable MapReduce Framework for All-Pair Similarity Joins of Multisets and Vectors. *PVLDB* 5, 8 (2012), 704–715.

[21] Fatemeh Nargesian, Erkang Zhu, Ken Q. Pu, and Renée J. Miller. 2018. Table Union Search on Open Data. *PVLDB* 11, 7 (2018), 813–825.

[22] Chuitian Rong, Chunbin Lin, Yasin N. Silva, Jianguo Wang, Wei Lu, and Xiaoyong Du. 2017. Fast and Scalable Distributed Set Similarity Joins for Big Data Analytics. In *ICDE*. 1059–1070.

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

[24] Akash Das Sarma, Yeye He, and Surajit Chaudhuri. 2014. ClusterJoin: A Similarity Joins Framework using Map-Reduce. *PVLDB* 7, 12 (2014), 1059–1070.

[25] Anshumali Shrivastava and Ping Li. 2015. Asymmetric Minwise Hashing for Indexing Binary Inner Products and Set Containment. In *WWW*. 981–991.

[26] Yasin N. Silva and Jason M. Reed. 2012. Exploiting MapReduce-based similarity joins. In *SIGMOD*. 693–696.

[27] Rares Vernica, Michael J. Carey, and Chen Li. 2010. Efficient parallel set-similarity joins using MapReduce. In *SIGMOD*. 495–506.

[28] Jin Wang, Guoliang Li, Dong Deng, Yong Zhang, and Jianhua Feng. 2015. Two birds with one stone: An efficient hierarchical framework for top-k and threshold-based string similarity search. In *ICDE*. 519–530.

[29] Jiannan Wang, Guoliang Li, and Jianhua Feng. 2012. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *SIGMOD*. 85–96.

[30] Xubo Wang, Lu Qin, Xuemin Lin, Ying Zhang, and Lijun Chang. 2017. Leveraging Set Relations in Exact Set Similarity Join. *PVLDB* 10, 9 (2017), 925–936.

[31] Chuan Xiao, Wei Wang, Xuemin Lin, and Haichuan Shang. 2009. Top-k Set Similarity Joins. In *ICDE*. 916–927.

[32] Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Efficient similarity joins for near duplicate detection. In *WWW*. 131–140.

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

[34] Erkang Zhu, Fatemeh Nargesian, Ken Q. Pu, and Renée J. Miller. 2016. LSH Ensemble: Internet-Scale Domain Search. *PVLDB* 9, 12 (2016), 1185–1196.

7 APPENDIX

7.1 Indexing Data Lakes

In this section, we discuss how we build an inverted index from a data lake in three major steps, implementation details, a strategy for incremental updates and how to process query in parallel.

7.1.1 Extracting Raw Tokens. In the first step, we extract sets from tables in the data lake. Each set is assigned a `SetID`, a unique integer identifier for the original table and column from which the set is extracted; Sets are flatterned into a mapping called `RawTokens` of tuples `(RawToken, SetID)`.

7.1.2 Building Token Table. In the second step, we build a mapping called `TokenTable`, that maps every token `RawToken` to a unique integer ID, `TokenID`, which indicates the token's position in the global order, and an integer duplicate group ID, `GroupID`. Each tuple of the mapping is `(RawToken, TokenID, GroupID)`.

To build `TokenTable`, we first build posting lists of sorted `SetID`, by grouping tuples in `RawTokens` by `RawToken`. Then we sort all posting lists by length (i.e., token frequency),

MurmurHash3 hash values[9], and the lists themselves. The use of MurmurHash3 hash helps us avoid most of expensive pair-wise comparisons of same-length posting lists that did not have hash collision. Then we use the sorted position of each posting list as the `TokenID` of the corresponding `RawToken` to obtain the mapping `(RawToken, TokenID)`. It is important to note that, even though we are using token frequency as the global order, our algorithm described in Section 3.4 works for any other global order, for example, the original positions of posting lists before sorting. In order to leverage the distinct list optimization described in Section 3.5, however, we must use frequency order.

As described in Section 3.5, each duplicate group spans a consecutive interval in the global order by frequency. To create duplicate groups, we must identify their starting and ending positions. The starting positions are indentified by by comparing every pair of adjancent posting lists (lower and upper) in the global order: for a pair, if the lower and upper posting lists are different, then the upper posting list's `TokenID` is the starting position of a new duplicate group. The first duplicate group's starting position is 0. Similary, the ending positions are identified by scanning every pair of adjacent groups' starting positions: the upper group's starting position is the ending position of the lower duplicate group. Once all the starting and ending positions are identified, we can generate the `GroupID` as the groups' positions in the sorted order, and the mapping `(TokenID, GroupID)` by enumerating from the starting to the ending position of every group.

Lastly in this step, we join the two mappings, `(RawToken, TokenID)` and `(TokenID, GroupID)` to obtain `TokenTable`.

7.1.3 Creating Integer Sets. The last step is to convert all tokens in sets to integers (`TokenID`). We use integers because computing intersection between integer sets is faster than between string sets, and storage systems such as Postgres[10] can read integer sets efficiently. Another benefit of using integer sets is better estimation of read cost, which is discussed in Section 7.2.

We create integer sets by first joining mappings `RawTokens` and `TokenTable` on `RawToken`, and then grouping by `SetID` to get sets while selecting only `TokenID` and `GroupID`. Each set is sorted by `TokenID` to reflect the global order. Lastly, we create the posting lists as shown in Section 2.4 from the integer sets.

7.1.4 Implementation and Performance. We implemented the indexing algorithm using Apache Spark[11]. The input is the `RawTokens` table of tuples `(RawToken, SetID)`, and the outputs are the integer sets and posting lists. Apache Spark automatically creates a series of tasks, some with inter-dependencies, for parallel execution on a cluster. Since Spark tasks can be scheduled one after another or simultaneously, the maximum per-task peak memory usage indicates the minimum amount of memory that must be available on each worker node before running into out-of-memory error. Table 4 lists the results on input and output sizes, duration, and maximum per-task peak memory.

Table 4: Indexing benchmark sets using Apache Spark on a cluster of 3 worker nodes each with 100 GB of memory and 63 cores.

	Open Data	Web Tables
Input `RawTokens`	37.3 GB	51.4 GB
Output Posting Lists	51.3 GB	36.0 GB
Output Integer Sets	10.6 GB	16.3 GB
Duration	2.4 h	1.1 h
Max Per-Task Peak Memory	7.1 GB	9.1 GB

7.1.5 Handling Incremental Updates. The posting lists and integer sets in our index can be updated incrementally. To do so we must maintain the global order of tokens to ensure the correctness of our algorithm as described in Section 3.4.

Any update to the index, such as adding a new set, a new token, or removing a token, can be expressed as a sequence of `ADD` and `DELETE` operations given `(RawToken, SetID)` tuples. For `ADD`, there are 4 different cases:

Case 1: `RawToken` and `SetID` both exist. This requires no action because a set consists of unique tokens.

Case 2: `RawToken` exists, and `SetID` is new. We create a new integer set using the existing `TokenID`. and then append to the posting list of `TokenID` a new entry $(\texttt{SetID}, 0, 1)$. See Section 2.4 for posting list entry. Due to the append, we invalidate the duplicate group ID of this posting list. The global order is unchanged.

Case 3: `RawToken` is new, and `SetID` exists. Because the token is not in our index, we assign a new integer `TokenID` by incrementing the maximum existing `TokenID`, effectively expending the global order by one and keeping the existing positions unchanged. Because the new token is at the end of the global order, we append its new `TokenID` to the end of the existing set given by the `SetID`. For existing posting lists of the set, we update the set's entry by incrementing the size: $(\texttt{SetID}, *, size + 1)$ – because the new token is appended, the existing tokens' positions are unchanged. Lastly, we create a new posting list for the new token with entry $(\texttt{SetID}, size, size + 1)$.

Case 4: `RawToken` and `SetID` are both new. This is the combination of the previous two cases. We first assign a new

[9]https://github.com/aappleby/smhasher
[10]https://www.postgresql.org/docs/10/static/intarray.html
[11]https://spark.apache.org

TokenID to the RawToken, then create a new integer set, and lastly create a new posting list with entry (SetID, 0, 1). The global order of tokens is expanded by one, but the rest is unchanged.

This incremental update strategy can also be applied directly to an empty index and the resulting global order would be the order in which tokens are added.

For DELETE, we first remove the entry from the posting list corresponding to RawToken, and update the entries in other posting lists of the tokens in the set: some positions need to be shifted by one, and the sizes are decremented by one. Then we also remove the token from the integer set.

In summary, the incremental update strategy keeps the global order of existing tokens unchanged, ensuring the correctness of our search algorithm (Section 3.4). The caveat is that we cannot assign duplicate group IDs for the newly added posting lists as well as those with append or removal, because finding the duplicate groups requires sorting all posting lists. Thus, we cannot skip the new and updated posting lists even if they are duplicates. This only affects the query performance, and since the tables in data lakes are often used for analytics tasks, updates are rare. The index can keep track of the number of posting lists affected and statistics on query runtimes, so that it can inform the system administrator to choose an appropriate time to rebuild the index for better performance.

7.1.6 Parallel Query Processing. A simple strategy to scale out the index is to randomly partition the sets into partitions, and build an index on each partition. Query processing is distributed across the partitions and the top-k results are merged to obtain the global top-k. To gracefully balance the partitions under updates, however, requires a more advanced solution. We will study it in our future work.

Within each partition, it is possible to utilize multi-core hardware to read and process postings lists in parallel. However, parallelizing reading candidates is more challenging, due to the k-th intersection size threshold for position filter pruning being mutable. It is possible to achieve parallism and improve running time if we allow less effective pruning. This problem warrants a future research work.

7.2 Estimating Costs

In this section we describe the functions $S(\cdot)$ and $L(\cdot)$ for computing the cost of reading sets and posting lists.

There are multiple components in the cost functions. First, there is *index access time* – the time to find where the set or posting list is located in the storage layer (e.g., disk, memory, etc.) given its pointer. For Open Data, since there are more than 563M posting lists, this time can be significant. Then, there is the time to read the set or posting list from the storage layer, and the time to transfer the data through network or inter-process communication, which involves serialization and deserialization. These components can be collectively called *read time*.

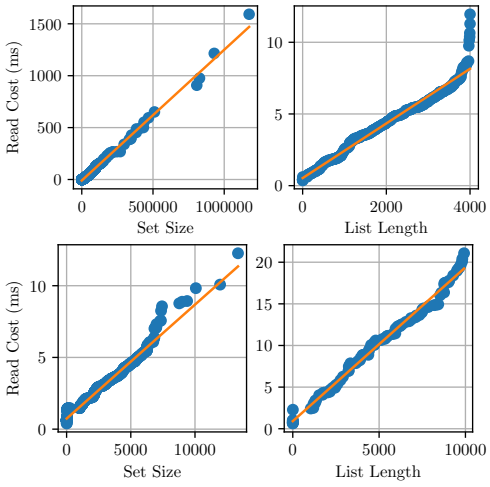

Figure 10: Samples of read costs of sets and posting lists in Open Data (top) and Web Tables (bottom), and fitted lines.

The index access time is likely a constant, especially with popular indexes such as B+ Tree used by many storage layers. On the other hand, the read time is proportional to the size of data, which needs to be transferred to the search engine process, regardless of the type of the storage system.

Thus, we use linear functions to express the read cost:

$$S(|X|) = s_0 + s_1 \cdot |X|,\ L(f_i) = l_0 + l_1 \cdot f_i \tag{15}$$

Where s_0 and l_0 are the index access times, and s_1 and l_1 are the factors for the read times.

In order to estimate the parameters in Equation 15, we sampled 1,000 sets and 1,000 posting lists and measure the I/O times, and then fit the functions to the sampled data points. Figure 10 shows the samples and fitted lines from sets and posting lists in Open Data and Web Tables, respectively.

7.3 Measuring Query Memory Footprint

Measuring memory footprint based on heap usage has to deal with noise such as overhead in programming language runtime, garbage collection, and various compiler optimizations. Thus, we only measure the amount of data structure allocated for the purpose of processing the query. In this section, we explain our approach to measure the query memory footprint of the algorithms in Section 4.4.

We use Equation 16 to calculate the memory footprint of MergeList-D. The first term is the size of a hash map for tracking the counts of times the candidates appear in posting lists. Its size is calculated using the number distinct sets encountered ($|W_{ML}|$), times the size of each set ID and count.

The second term is the size of the buffer that is allocated to read posting lists. We use the longest posting list read to calculate the size. Since `MergeList-D` only needs the set IDs in the each posting list, and does not need the token positions and set sizes, the buffer size is simply the total size of integer set IDs in the longest posting list.

$$|W_{ML}| \cdot 2 \cdot \textsc{SizeOf}(Int) + \max_{i \in [1, |Q|]} f_i \cdot \textsc{SizeOf}(Int) \tag{16}$$

We use Equation 17 to calculate the memory footprint of `ProbeSet-D`. The first term is the size of the buffer allocated for reading posting lists. Due to the use of prefix filter (subset of posting lists from 1 to p^*), the buffer size is the size of the longest posting list in the prefix. However, since `ProbeSet-D` needs the token positions and set sizes for position filter, in addition to set IDs, the size of each posting list entry is 3 integers. The second term is the size of the buffer allocated for reading sets. Similar to the other buffer, its size is calculated using the maximum size read. The third term is the size of the hash set allocated for tracking sets that have been pruned or read, so it is simply the size of the IDs of all the sets that appeared in the prefix posting lists. The set of all sets appeared in the prefix is W.

$$\begin{aligned} \max_{i \in [1, p^*]} 3f_i \cdot \textsc{SizeOf}(Int) + \max_{X \in W \setminus V} |X[j_{X,0} :]| \\ + |W| \cdot \textsc{SizeOf}(Int) \end{aligned} \tag{17}$$

Equation 18 is used to calculate the memory footprint for `JOSIE-D`, where δ is the extra posting lists read after the prefix filter, V^* is the set of pruned candidates using position filter, and W_i is the set of candidates after reading posting list i. See Equation 14 and 7 for their usages in running time analysis. The first three terms have nearly indentical expression as those of `ProbeSet-D`, however the magnitudes can be different, as $p^* + \delta \geq p^*$, and $W \setminus V^* \subseteq W \setminus V$. The last term is the maximum size of the hash map allocated for unread candidates (W_i) after reading a batch of posting lists ends at i. As described in Section 3.2, in addition to the set ID, we keep track of 4 additional integers for each candidate. Also, since pruned candidates are removed from the hash map, we use the maximum count of candidate sets during query processing to calculate the allocated size, as the deleted slots can be reused.

$$\begin{aligned} \max_{i \in [1, p^* + \delta]} 3f_i \cdot \textsc{SizeOf}(Int) + \max_{X \in W \setminus V^*} |X[j_{X,0} :]| \\ + |W| \cdot \textsc{SizeOf}(Int) + \max_{i \in [1, p^* + \delta]} 5|W_i| \cdot \textsc{SizeOf}(Int) \end{aligned} \tag{18}$$

For the LSH Ensemble algorithms, `LSHEnsemble-60` and `LSHEnsemble-90`, we use Equation 19. The first term is size of buffer allocated for reading sets, which is the maximum size over all sets read. The second term is the size of the hash set allocated for tracking the candidates retrieved from the LSH indexes.

$$\max_{X \in W_{LSH}} |X| + |W_{LSH}| \cdot \textsc{SizeOf}(Int) \tag{19}$$

During experiment, we record all the relevant variables needed to calculate memory footprint. The experimental results on Open Data and Web Table benchmarks are shown in Figure 5, 7, and 9. Please see Section 4 for detailed discussion.

7.4 Caching

The search algorithm itself does not deal with memory management or caching, which is handled by the storage layer which stores the posting lists and integer sets. The storage layer could be a separate process and its communications with the query prcessing process may involve serialization.

During the experiment, we used Postgres as the storage layer for posting lists and sets. In order to better reflect the worst case running time of the algorithms, we set the shared memory buffer size of Postgres to 128 MB, which is small enough compared to the data size (see Table 4) to avoid caching too many results.

In a practical deployment of the search engine, increasing the shared memory buffer size to utilize caching may be helpful, as it also depends on the choice of the storage layer, for example, a disk-based storage system would benefit but not an in-memory database.