Efficient set joins on similarity predicates

Sunita Sarawagi*
IIT Bombay
sunita@iitb.ac.in

Alok Kirpal†
IIT Bombay
alok@it.iitb.ac.in

ABSTRACT

In this paper we present an efficient, scalable and general algorithm for performing set joins on predicates involving various similarity measures like intersect size, Jaccard-coefficient, cosine similarity, and edit-distance. This expands the existing suite of algorithms for set joins on simpler predicates such as, set containment, equality and non-zero overlap. We start with a basic inverted index based probing method and add a sequence of optimizations that result in one to two orders of magnitude improvement in running time. The algorithm folds in a data partitioning strategy that can work efficiently with an index compressed to fit in any available amount of main memory. The optimizations used in our algorithm generalize to several weighted and unweighted measures of partial word overlap between sets.

1. INTRODUCTION

Modern database systems are increasingly used to store and handle set-valued attributes. Object relational DBMSs support sets as a data type and therefore need to process various kinds of queries on set-valued attributes. Text columns in a DBMS need to be viewed as sets of words for various kinds of retrieval and similarity queries. There is increasing research interest in better and efficient support for text data in relational databases driven by two emerging trends: integrating information retrieval (IR) functionality in databases and supporting semi-structured data formats like XML.

Although the storage of set-valued data is now commonplace, efficient support for performing joins on these is limited. Set joins can involve a variety of interesting predicates, however previous work has only concentrated on simpler forms like containment, equality or non-zero overlap [17, 18, 21, 15]. Real-life queries often need to pose more complex join predicates, such as,

- Overlap or intersect set size $> T$ where the goal is to find all sets that overlap in at least T words. We call this the T-overlap join, a special case of which is the earlier non-zero overlap join [17] with $T = 1$.
- Jaccard coefficient $> f$ where the goal is to find all set pairs where the ratio of the intersect size to union size is greater than a fraction f.
- Weighted match $> T$ where elements are attached with arbitrary weights (for example, inverse of their frequency in the database) and the goal is to find all set pairs where the total weight of the common/overlapping elements is $> T$.
- Cosine similarity $> f$ where each element is treated as a dimension, each set as a vector where the wth coordinate denotes the importance of element w in the set and the goal is to find all set pairs where the cosine of the angle between their respective vectors is $> f$

All of the above joins measure partial overlap between sets in various ways and select those with high overlap. These predicates are particularly valuable for text data processing where stricter predicates like total containment or equality are almost never useful. Applications like data cleaning and data integration, extensively rely on such joins for deduplicating records with text fields like names and addresses [22, 2, 16, 20, 11].

To the best of our knowledge ours is the first paper that presents efficient algorithms for efficiently processing joins on such predicates. Our goal is to return exact answers to these join predicates in contrast to previous work [8, 4, 7, 3, 12, 5] that concentrate on returning approximate answers where most but not all of the matching pairs are returned.

Our contribution. We propose an efficient algorithm called Probe_Cluster for computing similarity joins over set/text attributes. Normally, such similarity joins are expensive to compute for arbitrary values of thresholds and can lead to quadratic complexity of output size and computation time. We exploit the property that most join predicates will require records with high value of similarity and use the threshold to design algorithms that reduce the running time by as much as two orders of magnitude.

We show how to adapt existing memory intensive algorithms when the amount of available main memory is limited. Uniform partitioning of data is significantly more difficult when joins are based on *partial* overlap of sets as against complete containment or exact equality. Through experiments on real-life datasets we show that even as the amount

*Contact author
†Current affiliation: Yahoo Inc, Bangalore

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD 2004 June 13-18, 2004, Paris, France.
Copyright 2004 ACM 1-58113-859-8/04/06 . . . $5.00.

of memory is reduced by a factor of fifty, running time stays within a factor of 2.5 of the full memory version with our partitioning strategy.

We present a general framework for optimizing the evaluation of joins based on several kinds of thresholded similarity functions, including but not limited to, intersect size, Jaccard coefficient, cosine similarity and their weighted counterparts. We quantify the effectiveness and generalizability of this framework by showing that the running time for the same output size stays the same even when produced via highly varied similarity predicates.

Outline. We first review three existing algorithms from information retrieval, database and data mining literature, and show how these can be adapted for overlap-joins (Section 2). In Section 3 we propose a series of successive enhancements over existing algorithms. For each proposed enhancement, we report the reduction in running time. We prefer this layout to presenting a final algorithm and then reporting bulk numbers since it enables a better understanding of the reason and benefit of each feature of a new algorithm. None of the proposed algorithms handle arbitrarily large data set sizes since they rely on large in-memory datastructures. In Section 4 we adapt our optimized algorithm to work with limited memory. In presenting these algorithms we initially considered the weighted T-overlap join. In Section 5 we show how these can be extended for other partial overlap predicates. Related work appears in Section 6.

2. ALGORITHMS FOR SIMILARITY JOINS

We are given a finite universe U of elements or words $w_1, w_2 \dots w_W$, and a collection of records or set over U. Each word is associated with a given weight (default 1). Given a threshold T, our goal is to find *all* pairs of sets such that the total weight of common words between them is greater $\geq T$. We call this the T-overlap join. In this paper we will describe our algorithms assuming self-joins. The extension to non-self-joins is obvious.

2.1 Probe-Count

The Probe-Count algorithm is derived from the way keyword queries are answered during Information Retrieval [23] using an inverted index. The inverted index maps words to the list of record identifiers that contain that word. Such an index can be constructed in memory in one sequential scan of the data by inserting each scanned record into the record list associated with all words the record contains.

After constructing the inverted index, scan the data again. For each record r, probe the index using each word in r. This will yield a set of record lists corresponding to matching words. Each list l_w corresponding to word w of r is associated with a weight(w). Merge the record lists to find all records that appear in lists whose total weight $\geq$ the threshold T.

Merging of lists can be time-consuming since each list can contain a large number of record identifiers (RIDs). An established optimization is to create the index in such a way that the RIDs within a list are sorted. Then during merging we just need to maintain a frontier of the lists and at each step advance the frontier to the next RID which appears in lists whose total weight adds up to more than T. Such an RID can be found efficiently by using a heap to maintain the frontiers of all lists being merged. We then repeatedly pop the minimum RID from the heap, accumulate its weight if successive popped RIDs are the same, and push in the heap the next RID from the frontier of the popped list.

Let n_w denote the number of records containing word w and t denote the average number of words per record. The Probe_count algorithm requires $O(\sum_w n_w^2 \log(t))$ comparisons since a record list l_w associated with word w is selected for merging as many times as there are number of records in it, that is, n_w times. Each merge operation is roughly over t lists and since we are using a heap the merge time gets multiplied by $\log(t)$. The memory required by Probe_count to store the inverted index is $O(\sum_w n_w)$

2.2 Pair-Count

Another way to use the inverted index to find all T-overlap record pairs is suggested in [4] where the second step is to create pairs of RIDs within each record list in the inverted index. In a list l_w with n_w records and of weight(w), this will create $n_w(n_w - 1)/2$ items of the form (RID_1, RID_2, weight(w)). Aggregate the RID pairs over all lists to sum the total weight of each RID-pair. Finally, retain only pairs with total weight above the threshold T.

This will be a natural join plan when a set is expressed in the unnested representation where for each set-element pair we have an entry in a table. Then the above can be expressed as a self-join followed by group-bys to aggregate counts of each RID pair as show in [11, 12].

The Pair_count algorithm aggregates $\sum_w n_w^2$ pairs. We used a hash-function to do the aggregation, thus the time taken is $O(\sum_w n_w^2 H)$ where H is the time to hash a record pair. Estimating H is hard because in practice it does depend on the size of the entries inserted in the hash table due to collisions and other factors. The biggest problem of Pair_count is the huge amount of memory required to store all the distinct pairs over all the lists.

2.3 Word_Groups

An interesting variant is to map this problem to the well-known frequent itemset mining algorithms with the word-ids as the items and the RIDs as the transactions. The minimum support is two and the maximum itemset weight set to T. We can then use an efficient frequent itemset mining algorithm like Apriori [1] and FP-growth [13] to find all itemsets with total weight no less than T along with the list of RIDs that contain it. From each such list of RIDs, we output pairs of RIDs.

The main shortcoming of this method is that the groups of RIDs associated with an itemset can be overlapping. The same pair will be generated from all itemsets of weight T that can be formed out of the common words between the record pair. This is unnecessary since our goal is met by having *any one* group output a T-overlap record. For example, if a pair of records have $2T$ overlapping words they will appear in $C(2T, T)$ combinations of itemsets. A number of tricks can be used to reduce the blowup in the number of groups.

The first idea is to output small groups early even before the total weights of the items reaches T. An absolute support of 2 is otherwise too small for most frequent itemset mining algorithms. An itemset with support smaller than M records but greater than 2 is output and pruned from growing larger itemset. M can be set to a small number like 5.

The second idea is to periodically at each level, merge itemsets with highly overlapping sets of records. We cannot afford to compare all possible pairs of itemsets and explicitly check for overlap in RIDs. Instead we rely on hash functions that can bucket together sets that have high overlap. One such is the MinHash function that has already been used successfully in other applications [7, 4, 6]. The MinHash function provides a signature for a set of RIDs such that the probability that the signatures of two sets is equal is directly proportional to their overlap amount (measured as the ratio of the size of the intersection and the size of the union). The basic idea is to define a random order of the record identifiers (RIDs) and select the minimum index in the new ordering as the signature. This estimate can be made more accurate by selecting k signatures instead of one using k independent hash functions to define the random ordering. Thus, the fraction of common RIDs between two lists g_1 and g_2 is estimated as:

$$S(g_1, g_2) = \frac{|\{i|h_i(g_1) = h_i(g_2)\}|}{k}$$

where $h_i(g)$ is the MinHash on set g using the ith hash function.

We use the k MinHash signatures of each group to collapse all overlapping lists using the following algorithm. Treat each list g as a record associated with k words of the form $\{1.h_1(g), \ldots k.h_k(g)\}$. Merge lists that overlap in more than kp words where p is a parameter of the compaction algorithm. The Probe_Cluster algorithm described in Section 3.4 can be used to efficiently create such clusters in a single pass.

2.4 Experimental comparison

Our first step was to evaluate each of these three variants of existing algorithms for the purpose of performing the T-overlap join. We performed experiments on various kinds of sets obtained from two real-life datasets.

The Citation dataset consists of citation entries downloaded from CiteSeer by searching on the lastnames of the 100 most frequently referred authors. This gave us 250,000 citations of size 72 MB. The raw data had no underlying structure. We segmented the text record into five fields namely, author, title, year, page number and rest using CiteSeer's scripts.

The Address dataset consists of names and addresses obtained from various utilities and government offices of the city of Pune in India. The data had ten attributes: "lastname, firstname, middlename, Address1, $\cdots$ Address6 and Pin" and a total of 500,000 records of total size 40 MB.

From each of these datasets, we derived various kinds of sets, corresponding to common similarity functions that would arise when identifying duplicate entries in these lists. In Table 1 we show a list of four such functions along with the average number of elements per set and the total number of distinct elements (or words). The first function, All-words treats all words in the entire citation entry as the set whereas All-3grams forms sets out of each 3grams (sequence of three letters) in the citation, Similarly, we have two kinds of sets for the address dataset. Because of space limitation, in this paper we will only report graphs from experiments on the first function of each of the datasets, namely All-words from the citation database and All-3grams from the address dataset.

The experiments were performed on an IBM X220 server with 2 GB of main memory and dual Xeon processors rated at 1.1 GHz.

Function	Average set size	Number of elements
Citation dataset		
All-words	24	70000
All-3grams	127	29000
Address dataset		
All-3grams	47	37000
Name-3grams	16	14000

Table 1: Similarity functions along with the average size of the set over which they are computed and the total number of distinct elements over all sets in the data

Summary of experiments with initial algorithms. Our experiments with the above similarity functions and various values of threshold ranging from 90% to 20% of the average set size, showed that none of the three algorithms in the present form can process the whole of the citation or address dataset within a reasonable number of hours even for the highest threshold of 87%.

The Pair_count algorithm is not a viable option for even medium sized datasets. Even at 20,000 records the number of record pairs it generates does not fit in one gigabyte of main memory. Converting the algorithm to perform an external sorting/aggregation of the pairs is also not an option because the large intermittent number of pairs will lead to high disk read/write overheads. This algorithm was proposed in [4] for finding mirror pages where the special hash signature on word-grams was such that the number of entries per list was a small fraction of the database size. The signature used there is not guaranteed to return all possible predicates that match the threshold criteria. We believe that in most real-life dataset, where the goal is to find exact matches it will be hard to build hash functions that can avoid the problem of large lists arising from skewed word frequencies.

The Probe_count algorithm required significantly smaller memory sizes but it spent too much time in the list merge operation. Even for 50,000 records and the highest threshold of 80%, it required more than 90 minutes to complete. When run on the entire 250,000 records, the join did not complete in 16 hours! A well-known optimization is to remove stopwords, we will discuss this in the next section.

The Word_Group algorithm is competitive for small datasets but it suffers in terms of the amount of overlap it generates across groups resulting in large number of implied pairs. Although the compaction step reduces the amount of this overlap, significant time is wasted in unnecessary generation and compaction of groups. Also, with increasing number of records the memory requirement shoots up sharply. For example, at 50,000 records and a threshold of 73% ($T = 17$), the memory consumed was more than 2 GB after three levels of itemset generation. An FP-growth based implementation took much less memory but did not complete in two hours. None of these algorithms are meant to be run on such low support values.

3. OPTIMIZATIONS ON EXISTING ALGORITHMS

The above experimental evaluation found that all three algorithms are either too slow and/or require too much memory. We present a number of optimizations to make them more practical.

3.1 Optimized threshold sensitive list merge

The main bottleneck in the Probe_count algorithm is the time required to merge the RID lists obtained during the probe of the inverted index. As mentioned in Section 2.1, the established method of doing the merge is to insert the frontier of the sorted lists in a heap. Then repeatedly find the minimum and accumulate its weight if successive minimum values are the same.

We propose a mechanism to improve merge efficiency by exploiting the threshold T. Most real-life datasets follow an extremely skewed distribution of the frequency of occurrence of words and most of the time is spent in merging a few large lists. A known trick to handle skew in the IR community is to remove words with very high frequency of occurrence (these are called stop words). We can easily adapt this idea in computing T overlap joins using Probe_count by marking the top $T-1$ highest frequency words as stopwords and reducing the threshold for a record T by the number of stopwords it contains. We call this variant the Probe_stopWords algorithm.

We next propose a more gradual method of exploiting the threshold parameter T. Given a set of lists to be merged, we sort the lists in increasing order of size. Scanning from the large end, we select the largest set L of lists with total weight less than T. Let the remaining lists be S. A record that satisfies the threshold condition *must appear in at least one of the lists in* S. We therefore use the heap data structure to merge only the lists in S instead of all the lists. The advantage is that we do not waste time processing records that appear only in the lists in L. For each record returned during the merge, we perform a doubling binary search over each list in L in increasing order of size to accumulate the true match count. During this search within L, after each failed search, we check if the remaining lists are sufficient to meet the threshold condition for the current minimum record even if the record were to appear in all of them. If not, we terminate the search early and repeat for the next minimum record from the heap.

A sketch of the merge algorithm is given below.

We compare the running time of this optimized algorithm called Probe_optMerge with two baselines: the Probe_count algorithm of the previous section, and the Probe_stopWords algorithm (which is Probe_count with stop words removed as described earlier). In Figure 1 we plot against increasing database size and in Figure 2 we plot against increasing values of thresholds for a fixed database size. We find that while Probe_stopWords does provide some improvement beyond the basic Probe_count algorithm, the improvement obtained by Probe_optMerge is much higher. Running time reduces by a factor of five to hundred with the threshold optimization. For example, at a threshold of 21 (87% of average set size) on a 50000 record database, the running time reduced from 3000 seconds to 38 seconds — a factor of 80 reduction from Probe_count and a factor of 20 reduction from Probe_stopWords. Even for low values of thresholds (40% of average set size) the running time reduces by more than a factor of 5. Interestingly, the shapes of the two curves

Algorithm 1 MergeOpt(r, T, I)

1: Let $A = l_1, l_2, \ldots l_t$ be the record lists of index I in decreasing order of length corresponding to the t words $w_1 \ldots w_t$ of r
2: Compute cumulativeWt$(l_i) = \sum_{j=1}^{i}$ weight(w_j)
3: $L = l_1, l_2, \ldots, l_k$ such that k is the largest index for which cumulativeWt$(l_k) < T$
4: Insert frontiers of lists $S = A - L$ in a heap H.
5: **while** H not empty **do**
6: pop from H current minimum record m along with total weight $m.w$ of all lists in H where m appears
7: push in H next records from lists in S that popped.
8: **for** $i = k$ down to 1 **do**
9: if $(m.w+$ cumulativeWt$(l_i) < T)$ **exit-for**;
10: search for m in l_i using a doubling binary search method, and if found,
11: increment $m.w$ with weight(word(l_i)).

for changing threshold (Figure 2) clearly bring out the difference of the two algorithms in terms of their sensitivity to threshold. The running time of the optimized algorithm drops at a rate much sharper than linear as the threshold is increased.

Threshold optimization on Pair_count. We applied similar threshold based pruning to the Pair_count algorithms by not generating pairs from the longest set of lists L with total weight just less than T. After aggregating counts from the remaining lists S, we binary search for each record in a pair into the lists in L, terminating early using cumulative weights as described in algorithm MergeOpt. The performance improvement was not as significant as in Probe_count

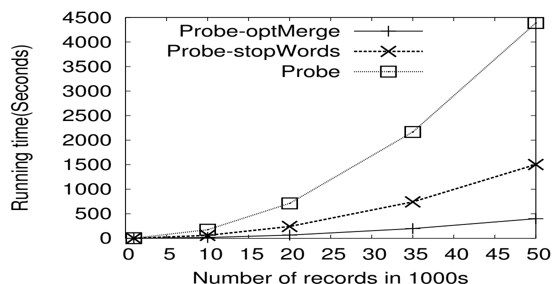

Figure 1: Running time versus dataset size averaged over all thresholds. (Citation data)

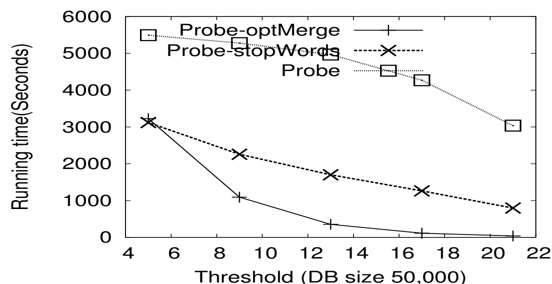

Figure 2: Running time versus threshold T for a fixed dataset size of 50000 records (Citation data)

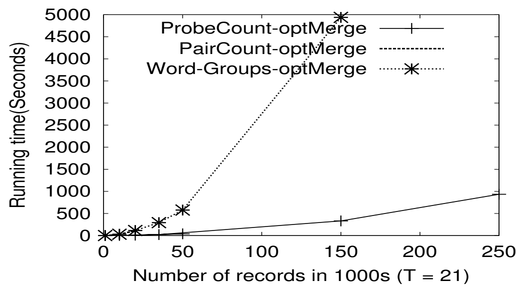

Figure 3: Citation data: Running time versus dataset size keeping T=21

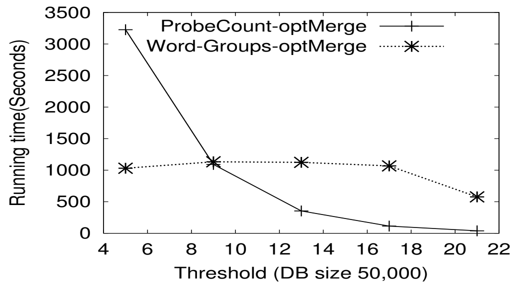

Figure 4: Citation data: Running time versus threshold T keeping dataset size = 50000

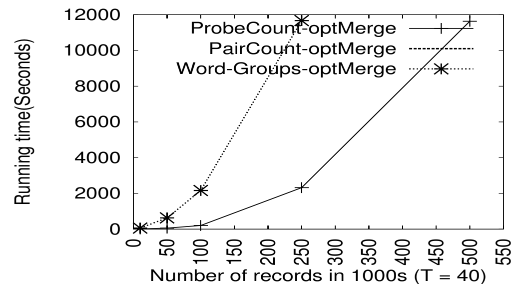

Figure 5: Address data: Running time versus dataset size keeping T=40

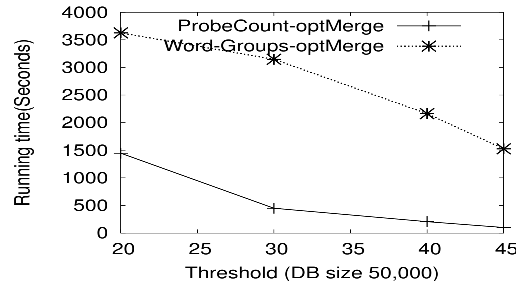

Figure 6: Address data: Running time versus threshold T keeping dataset size = 100000

and the memory requirement continued to be a problem because of the large number of pairs materialized. The optimized Pair_count algorithm could go upto 20,000 records (less than 10% of the total size of the citation dataset) on one gigabyte memory whereas the original one stopped at 10,000 records.

Threshold optimization on Word_Group. We exploit the threshold-based optimization in Word_Group as follows: using the set of large lists L as defined earlier, we modify the candidate generation phase so as to not generate wordgroups which only contains words in the list L. This is justified because the total weight of these words is $< T$. The average improvement in running time was only 20% with this optimization.

3.1.1 Experimental Evaluation

In Figures 3, 4, 5, and 6 we show a comparison of the threshold optimized Pair_count, Probe_count, and Word_Group algorithms for the citation and address datasets for changing threshold and database sizes. We found the optimized Probe_count algorithm to be superior by almost an order of magnitude to the other two algorithms. For example, at 150,000 records and $T = 21$ Probe_count took 5 minutes whereas Word_Group took 90 minutes. The only region where Word_Group is better than Probe_count is when the threshold T is equal to 5, which is 20% of the average record size. We do not expect joins with such low thresholds to be commonly deployed. For large absolute values of thresholds, for example $T = 45$ in the address dataset, the gap between Word_Group and Probe_count is higher because larger number of possible combinations implies more redundancy amongst the itemsets. The plots for the Pair_count algorithm are not visible in the graph because they only completed for very small dataset sizes (20,000) records.

These experiments clearly show that the only practical option for set overlap joins is Probe_count, both in terms of its memory and processing requirements. We propose a sequence of three further enhancements to the Probe_count algorithm. For each new enhancements we establish empirically how much it improves running time given all prior enhancement.

3.2 Single pass build and probe

The Probe_count algorithm decouples the process of index creation and probing performing each of these over two different passes. When performing self-joins these can be integrated so that in a single scan of the data as each record is scanned we first probe into the index, merge matching record lists and output record pairs. Finally, we insert the record in the inverted index. In addition to reducing data passes, a key advantage of this approach is that each probe step will be performed on the partial list rather than the full list. This simple optimization reduces the running time by a factor of two to three on an average as shown in the graphs in Figures 7 and 8 where ProbeCount_optMerge and ProbeCount_online denote the methods without and with online probe respectively. Joins on the full citation data which previously took three hours, now completed in a little over an hour. This range of improvement is also explained analytically since now the merge cost becomes proportional to $\sum_w n_w(n_w - 1)/2$ instead of $\sum_w n_w^2$.

3.3 Pre-sorting data

The merge time can be reduced even further by pre-sorting the records in *decreasing* order of the number of words in the record. This ensures that records with a large number of words get processed faster. Since the running time has

a $\log t$ factor, it helps to process long records (with large t) when the size of each RID-list in the index is smaller.

The sorting increases the number of scans but even after including this cost, we achieve upto a factor of two reduction in running time as shown in the graphs in Figures 7, 9, 8, and 10 where ProbeCount-sort indicates the method with the sort optimization. Previously, processing the 250,000 records took slightly over an hour. This time reduces by half even after including the time to do the sort.

3.4 Clustering related records

Another strategy to reduce the size of the record lists in the index, is to cluster together records with highly overlapping words and store in the index pointers to these clusters of records instead of individual records. The clusters are dynamically discovered and maintained in the same sequential pass that updates the index. Each cluster c is associated with a disjoint set of records and appears in the inverted index corresponding to all words that appear in any of the records in c. When a new record r arrives, we perform the usual probe-merge operation over the index and get back a list of clusters $C(r)$ each of whose union of words have T overlap with r. We need to follow this with a finer grained join over each record in the cluster. Each cluster maintains its own "word to record" inverted index. The record r then probes the indices of each cluster in $C(r)$ and outputs matching record pairs. The next step is to assign r to one of the existing clusters if there is sufficient similarity or, create a new cluster. Finally, the inverted index is updated so that all new words in r correctly point to the cluster of r. In Section 4 we will discuss details of the exact criteria used for assigning a record to a cluster.

The final algorithm that includes all four of the optimizations on Probe_count is called Probe_Cluster. Figures 7, 9, 8, and 10 show the running time of the final Probe_Cluster algorithm. The improvement obtained by this optimization will be highly sensitive to the number of high-overlap sets in the data. The citation dataset had lot more high-overlap sets than the address dataset. Hence we observe a greater improvement there.

The final Probe_Cluster algorithm is almost two orders of magnitude faster than the original Probe_count algorithm and more than an order of magnitude faster than the improve Probe_stopWords algorithm.

4. LIMITED MEMORY ALGORITHM

The inverted index used in Probe_count is of size proportional to the total words occurrences over all records. A wealth of techniques exist in IR [23, 19] for compressing an inverted index. These would contribute to pushing the limit upto which we can hold the index in memory. However, for large enough data sizes, the available memory will be incapable of holding even the compressed index. Then the only viable option left is to partition the data. We propose how in such cases we can effectively and efficiently partition data into groups of related records.

Unlike in equality joins using partitioned hashing where a record can be assigned to just one partition, for set overlap joins a record would typically have to be assigned to multiple other partitions. The partitioning method used in [21, 18] for handling set containment joins, is not applicable to our case since they heavily rely on the fact that *every* word in a joining record has to appear in the other record in a join.

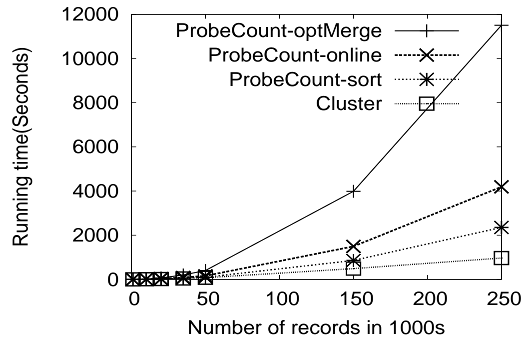

Figure 7: Running time versus dataset size averaged all thresholds (Citation data)

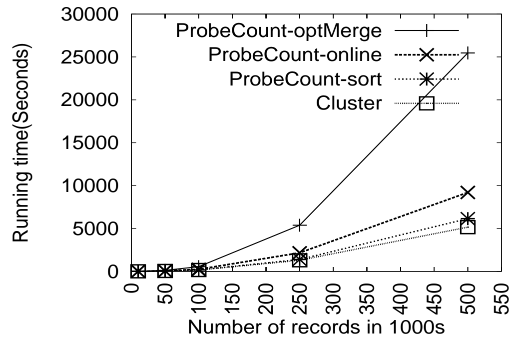

Figure 8: Running time versus dataset size(average all thresholds, Address data)

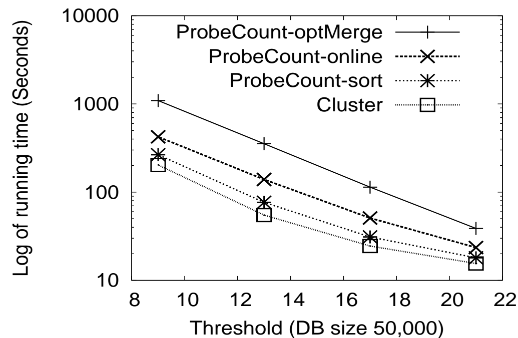

Figure 9: Running time versus threshold for dataset size = 50000 (Citation data)

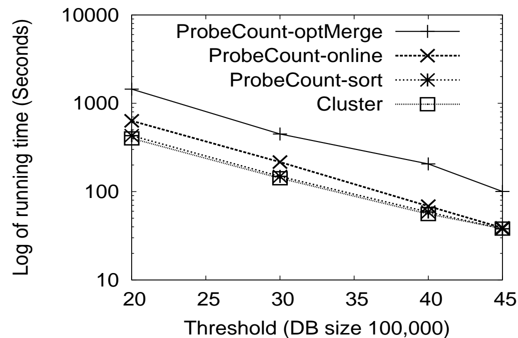

Figure 10: Running time versus threshold for dataset size = 100000 (Address data)

A good partitioning strategy should minimize the number of partitions per record without making any one partition large. Designing such an efficient partitioning strategy can be extremely challenging. Simple approaches like creating a different partition for each word, gave terrible performance for two reasons. First, partitions of frequent words could be almost as large as the size of the original data. Second, the amount of work repeated across partitions could be high since a record would appear in as many partitions as there are words in it. We can address the first problem by partitioning large lists further on words until each list is small enough. We can address the second problem by compacting together highly overlapping partitions. Both of these optimizations are available by doing the partitioning using the Word_Group algorithm of Section 2.3 with a minimum support of S where S is the largest number of records that can be held in memory. However, experiments showed that even this advanced method of partitioning gave bad performance.

We now present a method that avoids these problems by partitioning data on as large a set of words as possible. In the first phase, a compressed form of the inverted index is used to create these partitions of highly related records. The index can be compressed to fit in any available amount of memory, so that when sufficient memory is available, the method just reduces to the Probe_Cluster method discussed in section 3.4. Then, in the second phase we create finer grained indices over subsets of partitions that fit in memory and perform the joins. We elaborate on the two phases of the algorithm next.

4.1 First phase: data partitioning

Assume that in a preprocessing sequential pass over the data we have estimated various statistics like the total number of records N and the size of the full record-level inverted index in terms of the number of word occurrences W. Let the available memory be capable of holding only an index of size M and $M < W$.

In the first phase we construct a compressed inverted index of size no more than M. This index is simultaneously used to create partitions of the data. The inverted index can be compressed to fit in the available main memory in one of two ways:

1. Grouping together words with several overlapping records so that the number of word entries is reduced
2. Grouping together records with several overlapping words so that the length of the record list attached with a word is reduced

The first kind of groups on words can be created in one pass of the data by storing with each word a MinHash signature of the RIDs in that word (as discussed in Section 2.3) and then merging together related words. We observed that in most data sets although the number of words reduces sufficiently, this does not result in significant reduction in index size because the *larger lists* did not overlap enough. The error in merging unrelated large word lists, leads to bad partitioning decisions causing overall performance to deteriorate. We will therefore not discuss this method of compressing an index any further.

In contrast, the second kind of groups on records proved much more effective in compressing the index at varying levels of detail while retaining enough information to do good partitioning. We next elaborate on how these record clusters are created in an online manner.

Based on the statistics collected from the preprocessing step, we determine N_g the maximum number of clusters that the first pass inverted index can hold and N_R the maximum number of records that can be held per cluster. We estimate N_g as $\frac{N \times M}{W}$. Also, we fix $N_R = N_g$ under the assumption that $M \geq \sqrt{W}$. If this is violated, we can easily extend the algorithm to do recursive partitioning.

Now we scan the data sequentially. For each record r currently scanned we perform the following steps:

1. Find all clusters $J(r)$ with which word overlap is $\geq T$. This is the normal probe step of the Probe_Cluster algorithm. Record r needs to be joined with each cluster in $J(r)$ in the second stage.
2. Choose a home cluster $h(r)$ to which r should belong. The home cluster could be either an existing cluster with high enough word overlap and size $\leq N_R$ or, a new cluster provided the number of clusters is $\leq N_G$. Section 4.1.1 provides details of how this search for the home cluster is done efficiently without comparing a record explicitly to each existing cluster.
3. Insert in the index all new words in $h(r)$ due to the inclusion of r to $h(r)$.
4. Append to a disk store this record's partitioning information consisting of $r, J(r), h(r)$.

In this phase the only information maintained in memory is the compressed inverted index and a count of the number of records in each cluster. The output of this step is a file containing for each record, the set of other clusters with which it needs to be joined and a cluster to which it is assigned.

4.1.1 Finding the most similar cluster using the optimized inverted index

We define a similarity value between a record and a cluster and attempt to assign r to the most similar cluster. The challenge here is to find the best cluster without explicitly computing similarity with each of the existing clusters. We can define the similarity between a record r and a cluster c in a number of ways. Consider a simple definition as the number of common words between r and c.

We first show how with this definition we can efficiently adapt the probe-merge step of the inverted index to find the most similar record. The MergeOpt algorithm for merging strongly exploits the fact that we do not need to ever produce record pairs with overlap less than T. When memory is limited we might need to assign a record to a cluster even when the overlap is less than T. We show how to adapt the probe method without sacrificing its huge gain.

Initially, we start the probe with a very low threshold (say 20% of T), then as we find a matching cluster c of size $< N_R$ and overlap $O \leq T$, we increase the threshold to O. Dynamic increases of thresholds can be efficiently handled in MergeOpt because that just implies that some lists would be removed from the heap and put in the direct search list. Thus, each subsequent cluster returned by MergeOpt will have an overlap either greater than T or no less than the threshold of all previous clusters.

We showed how a single index probe can return all clusters with which r has to be joined and find the most similar cluster even if its overlap is $< T$.

We use a similarity function that is slightly better than overlapping word count. We compute similarity as the ratio of the overlap size to the union size in the two sets. This prevents large clusters from getting too large too fast. For this measure, we update the threshold to be the average of the previous threshold and the current overlap O when it is $\leq T$.

If the highest similarity of a record to the available clusters is lower than a similarity_threshold value and the maximum number of clusters is still not reached, we create a new cluster. We fix the similarity_threshold using a probability calculation derived from the average number of records desired per cluster and the average number of words per record. However, we do not go over the details of this estimation because of lack of space.

Algorithm 2 ClusterMem(D, T)

Estimate N_g, N_R in one sequential pass of data D
(optional) External sort data D by decreasing record length
Initialize index I and the cluster set Cs to be empty
{**First stage: data partitioning**}
for each record r in D scanned sequentially **do**
  Probe I to find clusters with increasing similarity to r and clusters $J(r)$ with $\geq T$ overlap. Choose from these the highest similarity cluster of size $< N_R$ as the home cluster $h(r)$.
  if similarity$(r, h(r)) <$ similarity_threshold and $|Cs| < N_g$ **then**
    create a new cluster $h(r)$ with r.
  Insert new words in $h(r)$ in I
  Append identifier of $r, h(r), J(r)$ to disk store pInfo.
{**Second stage: Finer joins**}
Partition Cs into batches $Cs_1 \dots Cs_k$ such that full index of clusters in each batch will fit in memory.
Divide entries in pInfo based on the k different partitions $\text{pInfo}_1 \dots \text{pInfo}_k$ on disk
for each partition Cs_i **do**
  Initialize index I_c for each cluster $c \in Cs_i$
  for each record entry r in pInfo_i **do**
    read record r from the database D
    join r with each $c \in J(r) \cap Cs_i$ by probing index of c
    if $h(r) \in Cs_i$, insert r in index of $h(r)$

4.2 Second phase: finer grained joins

In the second phase, we go over the partitioning information present on disk to efficiently perform the finer grained joins between records and clusters. For each cluster we need to create an in-memory inverted index over all records in this cluster and simultaneously probe it using all records with which it is supposed to join. We perform this in batches of clusters whose indices can fit together in memory.

After the first phase, we use the clustering information in memory to define these batches based purely on the total number of records in that cluster. Next we go over the partitioning file and break it up into batches that will be handled together. We go over each batch in turn, as a new record key is encountered we fetch the corresponding record from the database. For all clusters in the current batch with which it joins, we probe the index, perform the join and return the record pairs. Finally, if its home cluster is in the current batch, we insert it into the respective index.

We call this ClusterMem and is described in Algorithm 2.

Efficient disk I/O. The various file I/Os performed in the partitioning algorithm are optimized for sequential performance. When creating the partitions in the first phase, we perform only appends on the partition file pInfo. We store in pInfo only identifiers for records and clusters rather than the entire record. So, the file is not expected to be very large. In the second phase, this single file is partitioned across batches. Finally, for each batch the data records within a batch are fetched in the order in which they were scanned in the first phase. This makes the database probe efficient. We do not recommend storing the entire record in the partitioning file of each batch because unlike in normal hashing, the same record could still be spread across multiple batches. This might cause the database to be replicated across different batches.

4.3 Experimental evaluation

We use the datasets of Section 2.4 to show how our limited memory algorithm adapts to varying amounts of available main memory for the index. In Figure 11 we show the running time of ClusterMem with increasing amount of available memory M. For the citation dataset, as we decrease the amount of available memory by a factor of 5, running time increases only by a factor of 1.5. Further reducing memory by a factor of 50, increases running time only by a factor of 2. Similar behaviour is observed for the address dataset. For example, the full address dataset with 500,000 records and a threshold of 40 required storage of 20 million word occurrences in the inverted index and consumed about 2 gigabytes of memory when computed in a single pass. In the limited memory version with partitioning and the wordpairs restricted to 2% of the total word pairs, the memory consumed was less than 100 megabytes and the running time slightly over two times the one-pass version.

5. EXTENDING TO OTHER JOIN PREDICATES

We show how our final Probe_Cluster algorithm can be extended to handle join predicates other than those of the form "weighted set overlap $>$ a fixed threshold T". Examples of such functions are, Jaccard coefficient, cosine similarity on TF-IDF scores, and edit distance. The basic Probe_count algorithm without any of the optimizations is easily adapted to each case. The challenge is in making sure the various optimizations that lead upto the final Probe_Cluster algorithm can be exploited equally effectively for these other join predicates. In addition, we want to explore any new optimizations that are possible for these similarity predicates.

We cast the Probe_Cluster algorithm in a general framework that can be customized to the various similarity functions arising out of various ways of measuring partial overlap between the two sets. We present the general framework first and in Section 5.2 show how to customize these for different similarity functions. The framework can be defined in terms of the following three subroutines that will be specialized differently for each similarity predicate.

Word match score. The score that a match of a word contributes to the overall similarity of a record pair, is sometimes a function of the record instead of being a constant for

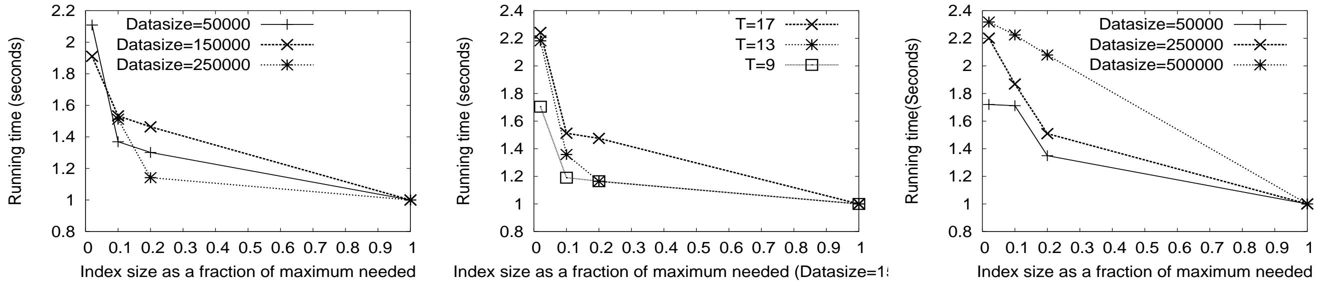

Figure 11: Running time versus index size for different data sizes and thresholds. The first two graphs are on the citation dataset and the last on the Address data.

each word. So in the general framework we have a scoring function of the form $\text{score}(w, r)$ for each word w, and record r. When a record pair (r, s) match on a word w_i we increment match amount by $\text{score}(w_i, r) \times \text{score}(w_i, s)$. Instead of a product of the scores any other function can also be easily supported. We assume a product for simplicity.

Threshold. In the general framework the threshold is a *function* of the scores attached with a record pair r, s instead of being restricted to be a constant value. We define the score of a record r as

$$||r|| = \sum_{w \in r} \text{score}(w_i, r)^2. \quad (1)$$

We allow the threshold function (denoted by $\mathsf{T}(r, s)$) to be any non-decreasing function of $||r||$ and $||s||$. Our goal then is to return all record pairs such that

$$\sum_{w \in (r \cap s)} score(w, r) \times score(w, s) \geq \mathsf{T}(r, s) \quad (2)$$

Additional Filters. Based on the form of the threshold and score functions above, specific similarity joins can identify additional filter predicates $\text{filter}(r, s)$ that can help eliminate pairs of records r, s *before* finding the set of words that are common between them. These filters can be applied in a number of ways. The simplest option is to apply the filter in the MergeOpt algorithm (Section 3.1) at the time a record is inserted in the heap H. We discuss other alternatives in Section 5.3.

5.1 Optimized Probe_count in the general framework

In this section we cast the optimizations that led up to the Probe_Cluster algorithm in terms of the above general framework.

5.1.1 Optimized list merging

We first show how we can adapt the MergeOpt algorithm of Section 3.1 to perform the threshold sensitive merge even when the threshold and word weights are not constants. The inverted index I now stores the $\text{score}(w, r)$ with each RID r in the list l_w corresponding to word w. The first step of MergeOpt that is affected is the splitting of the RID-lists into sets L and S such that any record s that meets the join condition of Equation 2 with r will appear in S and L will be as large as possible. For this we associate with each list l_w a $\text{score}(w_i, I)$ precomputed as:

$$\text{score}(w_i, I) = \max_{s \in I}(\text{score}(w_i, s)) \quad (3)$$

This score of each list can be maintained incrementally as the index is constructed since it does not depend on the probe record r.

We estimate the smallest possible threshold value with r on the current set of records in the index I as

$$\mathsf{T}(r, I) = \min_{s \in I} \mathsf{T}(r, s)$$

We can exploit the monotonic nature of $\mathsf{T}()$ and maintain with each index just a single minimum record score defined as $minS = \min_{s \in I} ||s||$. Thus, $\mathsf{T}(r, I) = \mathsf{T}(r, minS)$. Now, we can choose L as the largest set of lists which meet the condition $\sum_{w \in L} score(w, r) \times score(w, I) < T(r, I)$. The modified MergeOpt algorithm in terms of these generalized parameters is given next. Note that in the early termination step 9, we use the more accurate threshold $\mathsf{T}(r, m)$ instead of $\mathsf{T}(r, I)$.

Algorithm 3 MergeOptGen($r, \mathsf{T}(), I, \text{score}()$)

1: Let $A = l_1, l_2, \ldots l_t$ be the record lists of index I in decreasing order of length corresponding to the t words $w_1 \ldots w_t$ of r
2: $\text{cumulativeWt}(l_i) = \sum_{j=1}^{i} \text{score}(w_j, r)\text{score}(w_j, I)$
3: $L = l_1, l_2, \ldots, l_k$ such that k is the largest index for which $\text{cumulativeWt}(l_k) < \mathsf{T}(r, I)$
4: Insert frontiers of lists $S = A - L$ in a heap H.
5: **while** H not empty **do**
6: pop from H current minimum record m along with total score $m.w$ of all lists in H where m appears
7: apply $\text{filter}(r, n)$ before pushing in H a next record n from lists in S that popped.
8: **for** $i = k$ down to 1 **do**
9: if $(m.w + \text{cumulativeWt}(l_i) < \mathsf{T}(r, m))$ **exit-for**;
10: search for m in l_i using a doubling binary search method, and if found,
11: increment $m.w$ with $\text{score}(w_i, r)\text{score}(w_i, s)$.

5.1.2 Sorting criteria

In Section 3.3 we showed the benefits of pre-sorting records before insertion in the index such that longer records are processed earlier. In the general framework, the corresponding trick is to sort records in decreasing order of their score as defined in Eq 1. This order becomes all the more important

in the general case of non-constant thresholds because it implies that the threshold $\mathsf{T}(r, I)$ will remain high for a longer duration resulting in even better performance of MergeOpt.

5.1.3 Cluster summary when clustering related records

In Section 3.4 and Section 4 we proposed storing clusters of records in the inverted index. In the general case, we need to store additional statistics with each cluster so that if a record r joins with any record in a cluster C, then r will satisfy the join condition with C. With each cluster we store an overall score to be used for calculating thresholds as $||C|| = \min_{s \in C} ||s||$ and with each cluster-word pair, we store a value of score calculated over records in the cluster as

$$\text{score}(w, C) = \max_{s \in C} \text{score}(w, s).$$

5.2 Example predicates

We proceed by going over a list of alternative join predicates and for each discussing how to fit in the above optimization framework.

5.2.1 Jaccard coefficient

The Jaccard-coefficient expresses the similarity between two sets r and s as the ratio between the number of words in the intersection of r and s and the number of words in the union of r and s. The join predicate is of the form Jaccard-coefficient$(r, s) \geq f$ where f is a fractional value. We show how this predicate fits in the framework above.

Word match score. $\text{score}(w, s) = 1$.

Threshold. We need to find the largest $T(r, s)$ such that if $\text{Jaccard}(r, s) = \frac{|r \cap s|}{|r| + |s| - |r \cap s|} \geq f$, then $T(r, s) \leq |r \cap s|$. Rewriting we get

$$|r \cap s| \geq \frac{|r| + |s|}{1 + 1/f} = T(r, s)$$

Additional Filters. A filter condition that is applicable in this case is that the ratio of the number of words between the two sets should be $\geq f$, that is, we can filter all pairs r, s where $\min(\frac{|r|}{|s|}, \frac{|s|}{|r|}) < f$ before checking for common words between the two.

We can easily extend this to the weighted case where each word is associated with a weight and the intersection and union is on the weighted words.

5.2.2 Cosine similarity on TF-IDF scores

In this case, each word w in a record r has a weight that is inversely proportional to w's frequency in the input data corpus and directly proportional to its frequency in r. The exact function can take various forms: one popular variant is to express the TF-IDF score of a word w in a set r as

$$\text{TF-IDF}(w, r) = (1 + \log fr(w, r)) \log(1 + \frac{N}{fr(w)})$$

where N is the number of records, $fr(w)$ indicates the total frequency of the word over all sets. The motivation is to give more weightage to terms that occur rarely in the entire dataset but frequently in a set. This measure is popularly used in information retrieval for measuring the relevance of a document (viewed as a multiset of words) to a query (viewed as a set of words). Each document is then represented as a vector on words with the weight of the word forming the coordinate. The cosine of the angle between the two vectors is the similarity. Thus, the similarity between two documents s and r is defined as the dot product of the TF-IDF vectors divided by the norm of the vector for each set.

$$cosine(r, s) = \frac{\sum_w \text{TF-IDF}(w, r).\text{TF-IDF}(w, s)}{||r|| \; ||s||}$$

where norm of a vector

$$||s|| = \sqrt{\sum_w \text{TF-IDF}(w, s)^2}$$

The join predicate is of the form $cosine(r, s) \geq f$ where f is a fraction.

Word match score. In this case, the weight of each word is also dependent on the record in which it appears and is defined as

$$\text{score}(w, s) = \frac{\text{TF-IDF}(w, s)}{||s||}$$

Threshold. The threshold $T(r, s) = T(r, I) = f$ is independent of record pair.

For this function L will consist of lists that are both large and have low weightage. This is because the IDF scores of each word is inversely proportional to the number of records that contain it. Therefore, this optimization will be even more effective for TF-IDF matches than unweighted record match.

5.2.3 Edit-distance

Edit distance is hard to evaluate exactly without a quadratic comparison of pairs. However, as pointed in [11] there are several simpler conditions that all pairs within k edit distance should satisfy. For two strings r, s if edit-distance$(r, s) < k$ then, $|length(r) - length(s)| \leq k$, and if n_{12} denotes the number of matching q-grams between the two strings, then $n_{12} \geq max(length(r), length(s)) - 1 - q(k - 1)$.

The q-grams match predicate can be easily computed by treating each string as the set of q-qgrams in it and then performing a set-join.

Word match score. $\text{score}(w, s) = 1$.

Threshold. $T(r, s) = \max(length(r), length(s)) - 1 - q(k - 1)$ and $T(r, I) = T(r, m)$ where $m = \text{argmin}_{s \in I} \text{ length}(s)$, the shortest record.

Additional Filters. The filter condition that is applicable between two records r and s is that $|length(r) - length(s)| \leq k$.

5.3 Efficient evaluation of filters

All the filter conditions we presented above can be expressed as range conditions of the form $|l(r) - l(s)| \leq k$ where $l()$ is any ordered property of the record. In the case of Jaccard coefficient, $l(r)$ was log of the number of words in r and k was $\log(f)$. In the case of edit-distance, $l(r)$ was the length of the string.

The simplest option is to apply the filter in the MergeOpt algorithm (Section 3.1) at the time a record is inserted in the heap H. A second option is to range partition the records into possibly overlapping sets of records such that all record pairs that satisfy $|l(r)-l(s)| \leq k$ are together in at least one partition. Then invoke the Probe_Cluster algorithm on each partition. This option is only useful when the spread of l values is such that it is possible to create partitions without too much overlap.

The range filter can be thought of as a band join for which efficient algorithms exist in the database literature [10, 16, 20]. However, these methods produce as output pairs of records whereas our goal is to create large partitions on which we can invoke the T overlap join algorithms. We next present three algorithms for creating such partitions.

Simple. A simple algorithm is to first sort the records on $l()$. Then scan the sorted data to grow a window of records as long as the first record in the window is within range k of the current record scanned. If not, output the current window as a partition and mark the start of a new window by finding the next record in the window that is within range r of the current record.

This method outputs a huge number of overlapping records between adjacent windows. Often merging adjacent windows that have significant overlap is better even if it introduces some extra pairs that are outside the range constraint. We propose a greedy algorithm for this merge.

Greedy. In the above algorithm delay the output of a window w_{prev} until the following window w_{curr} is found. Then, merge the two adjacent window-groups w_{prev} and w_{curr} if that will lead to a smaller total join cost than leaving them as two separate partitions. This algorithm is not guaranteed to find the most compact partitioning.

Optimal. The optimal grouping can be found using a dynamic programming algorithm on the windows of the simple algorithm as follows: Let $w_1 \ldots w_n$ denote n adjacent overlapping windows. Draw a weighted graph with nodes consisting of each window and a start node w_0. The weight of edge (i, j) between w_i and w_j is the join cost of the partition formed by merging windows $w_{i+1} \ldots w_j$. The shortest path between nodes w_0 and w_n corresponds to the most efficient partitioning. This algorithm is computationally more expensive than the previous two but the better quality clusters lead to significantly reduced aggregation time.

In none of our datasets and similarity predicates did the partitioning option prove to be better than the simpler option of evaluating the filter at the time of list merging. However, this could be useful when the spread in the $l()$ values of records is very large.

5.4 Experimental evaluation

We establish that our generalization of the various join predicates is effective by performing the following experiments. We compare the running time of the different join predicates as a function of the number of matched pairs output for a fixed input database size. If the running time for a fixed size of output pairs is the same, irrespective of the join predicate used to produce it, then we have evidence that we are able to optimize these more complicated join predicates at least at the same level as the simpler fixed overlap predicate. Figures 12 show such graphs for three join predicates: intersect-size, Jaccard coefficient, and TF-IDF based cosine scores. Each graph was generated for a fixed size of the dataset and with changing values of threshold giving rise to different number of output pairs. We observe from the graph that the running times of the three functions are within a factor 20-30% of each other.

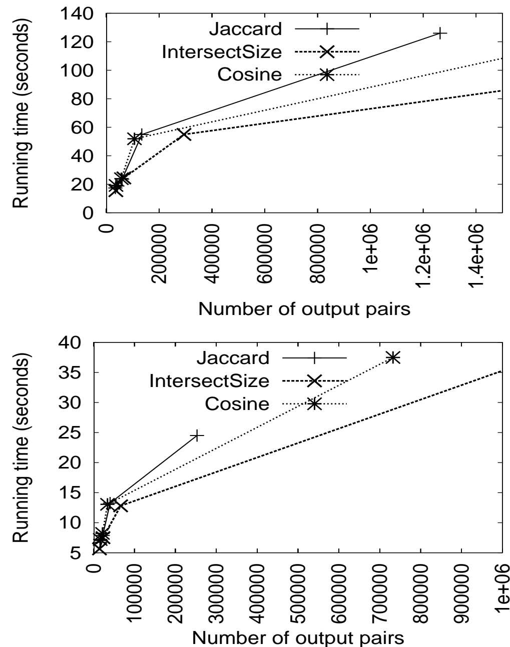

Figure 12: Running time as a fraction of output size for two different input data sizes: 50,000 (top) and 20,000 rows (bottom).

6. RELATED WORK

Existing algorithms for set joins in the database literature are limited to strict containment [21, 15, 18, 17], equality [17] and non-zero overlap joins [17]. The main techniques proposed in earlier work: signatures [15], partitioning [21] and adaptive partitioning [18] critically depend on the exact containment property and do not extend to partial overlap predicates.

There is extensive related work in the IR community on designing efficient methods for indexing and compressing text data [23] viewed as a set. The primary focus there is to efficiently answer keyword queries using optimized inverted indices. Our best performing algorithms are based on the same indexing method. Our contribution is in adapting these for joins for which related work is limited. For instance, the idea used in our MergeOpt algorithm for processing lists in increasing order of length is also deployed in IR when intersecting lists for answering conjunctive queries. Our contribution is in extending these to do list merging for T overlap matches, a problem harder than list intersection. The extensive techniques developed in IR for compressing an in-memory inverted index would be useful in our case too. The partitioning method we have developed are orthogonal to these compression techniques. There is also work in IR for constructing disk-resident inverted indices under limited

memory conditions [23, 14]. Some of these are based on data partitioning. However, the partitions in this case are very different from ours which are meant to group related records together for similarity joins.

In this paper we have concentrated on returning exact join results. Several previous work [8, 4, 7, 3, 12, 5], have concentrated on the problem of returning approximate answers to such similarity functions. Approximate methods make it easy to apply various sampling (as in [8] and [12]) and signature techniques (like MinHash in [4, 7, 3]). These are not applicable directly to the exact case. Signature-based methods have been evaluated vis-a-vis indexing methods for simpler exact joins like containment, equality and non-zero overlap joins in [17] and have not been found to be effective. In addition, signatures are harder to design for exact T overlap joins of the variety we address in this paper. Another form of similarity join is addressed in [9] where the goal is to return the top r most similar record pairs based on TF-IDF match. The idea there is to use an A^* search driven by bounds on scores derived from TF-IDFs weights. Although the details are different, the early termination and split strategies used in MergeOpt algorithm bear resemblance to the A^* search.

7. CONCLUSION

In this paper we presented an efficient algorithm for joining sets based on various similarity predicates. Starting from existing indexing methods in the IR literature we have developed a practical algorithm by incorporating a number of optimizations. Each optimization is individually analyzed for its usefulness in improving running time. The most significant of these was the threshold sensitive list merge procedure. This optimization alone was responsible for one to two orders of magnitude improvement in running time.

We propose a method of clustering records online so as to make the algorithm adapt to limited amount of memory. There are two interesting ideas in this part: (1) the way the index is searched with increasing threshold to efficiently return the most similar cluster, and, (2) the partitioning method that avoids the problems of data skew and redundant computation while creating clusters based on similarity along multiple words rather than single words. Even as the amount of memory is reduced by a factor of 50 running time stays within a factor of 2.5 with our method

We cast the algorithm in a general framework that makes it possible to easily exploit all of these optimizations to several other non-trivial joins like Jaccard coefficient and cosine similarity.

Acknowledgements. We would like to thank our funding agencies: Naval Research Board, New Delhi and Microsoft Research, Redmond for their generous support.

REFERENCES

[1] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. In *Proc. of the 20th Int'l Conference on Very Large Databases*, Santiago, Chile, September 1994.

[2] R. Ananthakrishna, S. chaudhuri, and V. Ganti. Eliminating fuzzy duplicates in data warehouses. In *VLDB*, 2002.

[3] Broder. On the resemblance and containment of documents. In *SEQS: Sequences '91*, 1998.

[4] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. In *Proc. Sixth Int'l. World Wide Web Conference*, pages 391–404. WWW Consortium, 1997.

[5] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD*, 2003.

[6] Z. Chen, F. Korn, N. Koudas, and S. Muthukrishnan. Selectivity estimation for boolean queries. In *Symposium on Principles of Database Systems*, pages 216–225, 2000.

[7] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J. D. Ullman, and C. Yang. Finding interesting associations without support pruning. *Knowledge and Data Engineering*, 13(1):64–78, 2001.

[8] E. Cohen and D. Lewis. Approximating matrix multiplication for pattern recognition tasks. *J. Alg. special issue of selected papers from SODA '97.*, 30:211–252, 1999.

[9] W. Cohen. Data integration using similarity joins and a word-based information representation language. *ACM Transactions on Information Systems*, 18:288–321, 2000.

[10] D. DeWitt, J. Naughton, and D. Schneider. An evaluation of non-equijoin algorithms. In *Proceedings of the 17th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Barcelona*, 1991.

[11] L. Gravano, P. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *Proc. of the 27th Int'l Conference on Very Large Databases (VLDB)*, Rome,Italy, 2001.

[12] L. Gravano, P. G. Ipeirotis, N. Koudas, and D. Srivastava. Text joins for data cleansing and integration in an RDBMS. In *ICDE*, pages 729–731, 2003.

[13] J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Mining and Knowledge Discovery*, 2003.

[14] S. Heinz and J. Zobel. Efficient single-pass index construction for text databases. *Journal of the American Society of Information Science and Technology*, 54(8):713–729, 2003.

[15] S. Helmer and G. Moerkotte. Evaluation of main memory join algorithms for joins with set comparison join predicates. In *The VLDB Journal*, pages 386–395, 1997.

[16] M. A. Hernandez and S. J. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Mining and Knowledge Discovery*, 2(1):9–37, 1998.

[17] N. Mamoulis. Efficient processing of joins on set-valued attributes. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 157–168. ACM Press, 2003.

[18] S. Melnik and H. Garcia-Molina. Adaptive algorithms for set containment joins. Technical report, Stanford University, 2001.

[19] A. Moffat, J. Zobel, and N. Sharman. Text compression for dynamic document databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(2):302–313, March-April 1997.

[20] A. E. Monge and C. P. Elkan. The field matching problem: Algorithms and applications. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96)*, 1996.

[21] K. Ramasamy, J. M. Patel, J. F. Naughton, and R. Kaushik. Set containment joins: The good, the bad and the ugly. In *VLDB*, 2000.

[22] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *Proc. of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining(KDD-2002)*, Edmonton, Canada, July 2002.

[23] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann Publishing, San Francisco, 1999.