Weighted Proximity Best-Joins for Information Retrieval†

Risi Thonangi,[1] Hao He,[2] AnHai Doan,[3] Haixun Wang,[4] Jun Yang[1]
[1]*Department of Computer Science, Duke University;* {rvt,junyang}@cs.duke.edu
[2]*Google Inc.;* haohe@google.com
[3]*Department of Computer Science, University of Wisconsin;* anhai@cs.wisc.edu
[4]*IBM T. J. Watson Research Center;* haixun@us.ibm.com

***Abstract*— We consider the problem of efficiently computing weighted proximity best-joins over multiple lists, with applications in information retrieval and extraction. We are given a multi-term query, and for each query term, a list of all its matches with scores, sorted by locations. The problem is to find the overall best matchset, consisting of one match from each list, such that the combined score according to a scoring function is maximized. We study three types of functions that consider both individual match scores and proximity of match locations in scoring a matchset. We present algorithms that exploit the properties of the scoring functions in order to achieve time complexities linear in the size of the match lists. Experiments show that these algorithms greatly outperform the naive algorithm based on taking the cross product of all match lists. Finally, we extend our algorithms for an alternative problem definition applicable to information extraction, where we need to find all good matchsets in a document.**

I. Introduction

Information retrieval today has gone far beyond finding documents with matching keywords. Many systems have significantly broadened the concept of a "match." For example, commercial search engine offerings, such as *Powerset* (www.powerset.com) and *AskMeNow* (www.askmenow.com), are able to handle questions such as "who invented dental floss," which cannot be answered by simply matching words in a document with "who" in a black-or-white fashion. Recent work from academia, by Chakrabarti et al. [7] and Cheng et al. [8], has made significant inroads into *question answering* and *entity search* (as opposed to document search). Critical to their success is the joint consideration of the qualities of "fuzzy" matches and the proximity among the matches.

To illustrate this approach, suppose we are interested in finding partnerships between PC makers and sports. A user may formulate this question as a three-term query: {"PC maker," "sports," "partnership"}. Figure 1 shows a sample document. While the document is obviously relevant, we want to go a step further—respond to the question directly with an answer, e.g.: Lenovo partners with NBA.

Simple keyword matching is clearly not enough to obtain good answers. The document does not mention the word "sports," but with additional background knowledge about sporting events and organizations, we can match "NBA," "Olympic Games," etc. As for "PC maker," there is in fact an exact match, but it does not help in answering the question. With the knowledge of which companies are PC makers, we can also match "Lenovo," "Dell," etc. We can match "laptop maker" too, if we know that laptops and PCs are closely related concepts. Finally, "partnership" matches with not only "partner" and "partnership," but also "deal" (though not as perfectly). Note that the matches are naturally weighted (or scored) by quality, as measured by how closely they relate to the query terms, or how confident we are that they correspond to the user's intentions. For scoring individual matches, a variety of techniques exist, including natural language processing, ontology, knowledge bases, named entity recognizers, etc.

Besides the individual match scores, another important factor considered by [7, 8] in assessing an answer is the proximity among the matches that constitute the answer. Intuitively, we are more confident in matches that are close together within the document. For example, in Figure 1, one would guess that {"Lenovo," "NBA," "partner"} have much tighter association than {"Hewlett-Packard," "Olympic Games," "partnership"}.

Ideally, we would like to find a set of matches, one for each query term, with high individual scores and close proximity to each other. This operation is a natural and important primitive in systems that jointly consider individual match scores and proximity among matches.

Algorithmic Efficiency Algorithmically, finding the highest-scoring answers within a document in this setting can be thought of as a weighted proximity "best-join" over lists. The input to the problem is a set of *match lists*, one for each query term, which contains all matches for the term in a document. Each match has two attributes: a location within the document, and a score measuring the quality of the match with respect to the query term. We join together matches across lists to form answer *matchsets*. Figure 1 illustrates the concepts.

A scoring function is used to combine individual match scores (***weight***s) and the ***proximity*** of match locations in order to score a matchset. We are then interested in identifying the ***best*** (highest-scoring) matchsets in the document—hence the name ***weighted proximity best-join***.

Various functions have been proposed in the literature. For

†The first, second, and last authors were supported by an NSF CAREER award (IIS-0238386) and an IBM Faculty Award. The third author was supported by an NSF Career Award (IIS-0347903), an Alfred Sloan fellowship, an IBM Faculty Award, and grants from Yahoo! and Microsoft.

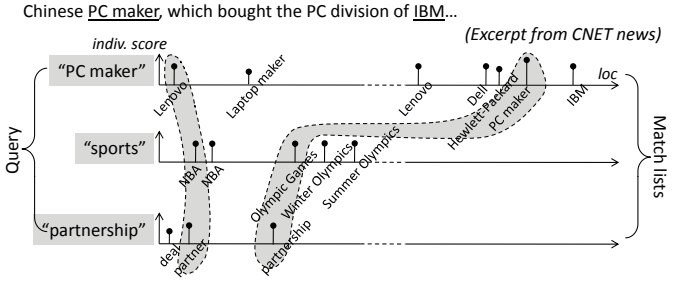

Fig. 1. An example illustrating our problem. Individual matches (underlined in the accompanying text) are shown as points whose x-coordinates correspond to match locations and y-coordinates correspond to individual match scores. Two matchsets (out of many other possible ones) are circled.

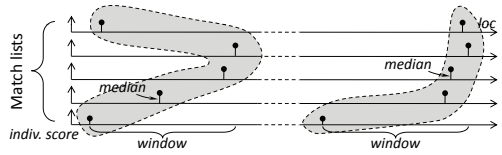

Fig. 2. Two matchsets with different degrees of clusteredness but equal-size enclosing windows.

example, Chakrabarti et al. [7] handle questions involving one "type" term (such as "who" or "physicist") and regular keyword terms. They decay the score of a match for the type term over its distance to the matches for other terms. Cheng et al. [8] consider queries with a general mix of "entity" types and regular keyword terms. Within a document, each matchset is scored by the product of the individual match scores, multiplied by a decreasing function of the length of the smallest window containing all matches.

Much work has gone into demonstrating the potential of combining individual match scores and proximity, and studying what matchset scoring functions produce the most meaningful answers. However, few have considered the efficiency of finding the best matchsets. A naive algorithm for finding the best matchset in a document would enumerate the cross product of all lists, evaluate the matchset scoring function for every possible matchset, and then return the one with the highest matchset score. This approach can be quite expensive. The number of matches per list could be substantial, especially since we look beyond exact keyword matches and include fuzzy matches. The size of the cross product could be exponential in the number of terms in the query, with basis of the exponent being the average size of the match lists; even a few query terms can blow up the running time dramatically.

Contributions In this paper, we focus on developing efficient algorithms for finding high-scoring matchsets under different scoring functions. Specifically, we make the following contributions.

First, we formalize the weighted proximity best-join problem and consider three types of scoring functions, *window-length*, *distance-from-median*, and *maximize-over-location*. Inspired by the scoring function from [8], window-length scoring functions incorporate proximity by decaying the matchset score with the length of the smallest window enclosing all matches in the matchset. While simple and intuitive, window length alone is not always enough. For example, although the second matchset in Figure 2 is intuitively much better than the first one, the scoring function fails to distinguish them because their smallest enclosing windows are of the same length. The other two scoring functions we consider overcome this limitation: distance-from-median has a form that can better capture the notion of proximity; maximize-over-location provides an even tighter coupling of proximity and individual match scores.

Second, we propose algorithms for computing the overall best matchset in a document under the three types of scoring functions, exploiting their respective properties for efficiency. Despite the flexibility in our scoring function definitions, our algorithms maintain good performance guarantees (that running time is linear in the total size of match lists), and substantially outperform the naive algorithm based on cross product. These strong performance results make the proposed scoring functions and associated algorithms practical additions to the information retrieval toolbox.

Finally, although for questions such as "who invented dental floss," finding one best matchset within each document is sufficient, it is not enough for some applications. For instance, returning to the example of Figure 1, we might want to extract all good matchsets for the query from the document. These include not only {"Lenovo," "NBA," "partner"}, but also {"Lenovo," "Olympic Games," "partnership"}, etc. Such a need for extracting all good matchsets often arise in information extraction applications. We consider an alternative problem definition that finds all matchsets that are "locally" best (with respect to different locations within the document), which can be further filtered to return matchsets with good enough scores. We show how to modify our algorithms to accomplish this new task while maintaining their linear complexities in terms of the total size of the match lists.

II. Preliminaries

Definition 1. *A query Q consists of a set of* query terms $q_1, q_2, \ldots, q_{|Q|}$. *Given a document, for each query term q_j, a* match list L_j *is a list containing all* matches *for q_j in the document, where each match m has a* location $\mathrm{loc}(m) \in \mathbb{N}$, *and a* score $\mathrm{score}(m, q_j) \in \mathbb{R}$. *Matches in each list are sorted in increasing order of their locations.*

A matchset M *for query Q consists of $|Q|$ matches $m_1, m_2, \ldots, m_{|Q|}$, where each m_j is a match for query term q_j (i.e., $m_j \in L_j$). A* (matchset) scoring function *computes the* score of matchset M *with respect to query Q, denoted* $\mathrm{score}(M, Q)$, *as a function over* $\mathrm{loc}(m_j)$ *and* $\mathrm{score}(m_j, q_j)$ *for all* $j \in [1, |Q|]$.

In this paper, we assume that match lists (and the individual match scores) are given. In practice, depending on the system and application scenario, match lists can be either computed online, by scanning an input document and matching tokens

against query terms, or derived from precomputed inverted lists.[1] Typically, the match lists are sorted in the increasing order of match locations; therefore, we only assume that they can be accessed in a sequential fashion.

In Sections III–V, we present three types of matchset scoring functions and associated algorithms for finding an overall best matchset (with the highest score) for a query Q from its match lists. The problem is formalized below.

Definition 2 (Overall-Best-Matchset Problem)**.** *Given query Q and associated match lists $L_1, \ldots, L_{|Q|}$, the* overall-best-matchset problem *finds a matchset with highest score, i.e.,*

$$\arg\max_{M \in L_1 \times L_2 \times \cdots \times L_{|Q|}} \text{score}(M, Q).$$

As briefly discussed in Section I, a naive solution to the overall-best-matchset problem is to consider all possible matchsets (i.e., the cross product of all match lists), compute their scores, and pick one with the highest score. The time complexity is $\Theta(|Q| \prod_{j=1}^{|Q|} |L_j|)$, which will be slow if there are more than just a couple of terms or some large match lists. Our goal is to develop better solutions whose complexities are linear in the total size of all match lists.

Finally, as noted in Section I, different applications may find variations and refinements of the overall-best-matchset problem more appropriate for their needs. We discuss how to extend our algorithm to handle these cases in Section VII.

III. Window-Length (WIN) Scoring

As discussed in Section I, a natural way of scoring a matchset is to add or multiply the individual match scores together, and then penalize the result score by the length of the smallest window containing all matches in the matchset. We formalize this type of scoring functions below.

Definition 3 (Window-Length (WIN) Scoring Function)**.** *Given a query Q and a matchset $M = \{m_1, \ldots, m_{|Q|}\}$, the* window-length (WIN) scoring function *has the following form:* $\text{score}_{WIN}(M, Q) \stackrel{def}{=}$

$$f\Big(\sum_j g_j(\text{score}(m_j, q_j)),\ \max_j(\text{loc}(m_j)) - \min_j(\text{loc}(m_j))\Big),$$

where:

- *g_j ($1 \le j \le |Q|$) are monotonically increasing functions.*
- *$f(x, y) : \mathbb{R}^+ \times \mathbb{R}^+ \to \mathbb{R}$ is monotonically increasing in x and monotonically decreasing in y; i.e.,*

$$\forall y : \big(x \ge x'\big) \to \big(f(x, y) \ge f(x', y)\big);$$
$$\forall x : \big(y \ge y'\big) \to \big(f(x, y) \le f(x, y')\big).$$

- *f satisfies the* optimal substructure *property; i.e., $\forall \delta \ge 0$,*

$$f(x, y) \ge f(x', y') \to f(x + \delta, y) \ge f(x' + \delta, y');$$
$$f(x, y) \ge f(x', y') \to f(x, y + \delta) \ge f(x', y' + \delta).$$

[1]Cheng et al. [8] propose precomputing inverted lists for entity types. Alternatively, a match list for a general concept (e.g., "PC maker") can be obtained by merging inverted lists of specific terms (e.g., "Lenovo," "Dell," etc.). Chakrabarti et al. [7] take a hybrid approach.

We have intentionally left functions f and g_j's as unspecified as possible. Specific choices of f and g_j's depend on the application, and are beyond the scope of this paper. To help make the definition more concrete, however, consider the following scoring function, which approximates the one used by Cheng et al. [8] by replacing their empirically measured distance-decay function with exponential decay (and ignoring their order and adjacency constraints):

$$\Big(\prod_j \text{score}(m_j, q_j)\Big) \times e^{-\alpha(\max_j(\text{loc}(m_j)) - \min_j(\text{loc}(m_j)))}, \quad (1)$$

where $\alpha > 0$. Exponential decay is a common choice for distance-decay functions (e.g., in TeXQuery [3] and ObjectRank [4]). The empirically measured distance-decay functions in [7, 8], although somewhat jagged, also resemble exponential decays. Clearly, (1) is a WIN scoring function, where $g_j(x) = \ln(x)$ is monotonically increasing and $f(x, y) = \exp(x - \alpha y)$ is monotonically increasing in x, monotonically decreasing in y, and satisfies the optimal substructure property.

Algorithm We give an algorithm that works for any WIN scoring function as long as f satisfies the properties prescribed in Definition 3.

The algorithm is based on dynamic programming. It examines all match lists in parallel, processing matches one at a time in the increasing order of their locations. Let $m^{(i)}$ denote the i-th match examined by the algorithm, and let $l^{(i)} = \text{loc}(m^{(i)})$. At $l^{(i)}$, the algorithm finds the best *partial matchsets* at $l^{(i)}$, formally defined as follows.

Definition 4 (Partial Matchsets)**.** *A* (partial) *P-matchset at location l, where $P \subseteq Q$ and $P \neq \varnothing$, consists of $|P|$ matches, one for each query term in P, all located at or before l. The (WIN) score of a P-matchset M_P at location l, denoted $s(M_P, l)$, is defined as:*

$$f\Big(\sum_{m_j \in M_P} g_j(\text{score}(m_j, q_j)),\ \ l - \min_{m_j \in M_P}(\text{loc}(m_j))\Big). \quad (2)$$

A best P-matchset at l is one that maximizes $s(M_P, l)$.

Note that an overall best matchset M must be a best Q-matchset at the last location of matches in M. Therefore, to find the an overall best matchset, the algorithm can find a best Q-matchset at each possible match location, and return the matchset with the highest score after processing all matches.

Now, at the i-th match $m^{(i)}$, how does the algorithm find a best P-matchset at $l^{(i)}$? We show that it can be computed from the best partial matchsets at the previous match location, $l^{(i-1)}$. Let $q^{(i)}$ be the search term that $m^{(i)}$ matches. The set of all P-matchsets at $l^{(i)}$ can be divided into two groups: those that contain $m^{(i)}$ and those that do not.

First, consider $\mathcal{M}_1$, the group of P-matchsets at $l^{(i)}$ that do not contain $m^{(i)}$. We claim that a best P-matchset at $l^{(i-1)}$ would also be best among $\mathcal{M}_1$ at $l^{(i)}$. The reason is that $\mathcal{M}_1$ is the same as the set of P-matchsets at $l^{(i-1)}$. Their partial matchset scores are affected only by increasing l in (2) by $l^{(i)} - l^{(i-1)}$. By the optimal substructure property of f, a best P-matchset at $l^{(i-1)}$ remains best among $\mathcal{M}_1$ at $l^{(i)}$.

Second, consider $\mathcal{M}_2$, the group of P-matchsets at $l^{(i)}$ that contain $m^{(i)}$ (this group would be empty if $q^{(i)} \notin P$). In

Algorithm 1: Computing overall best matchset for WIN.

```
1  MaxJoinWIN(Q, L_1, ..., L_|Q|) begin
2    M ← ⊥; S ← ⊥;    // M: overall best matchset found so far; S: its score
3    foreach nonempty P ⊆ Q do
4      M_P ← ⊥;
5      g_P^Σ ← ⊥; l_P^min ← ⊥;    // score components for incremental computation
6    foreach match m ∈ L_1 ∪ ··· ∪ L_|Q| in location order do
7      q_j ← the query term that m matches;
8      g ← g_j(score(m, q_j)); l ← loc(m);
9      foreach nonempty P ⊆ Q in decreasing sizes do
10       if {q_j} = P then
11         if M_P = ⊥ or f(g_P^Σ, l − l_P^min) < f(g, 0) then
12           M_P = {m};                    // found best single-term matchset at l
13           g_P^Σ ← g; l_P^min ← l;
14       else if q_j ∈ P then
15         if M_{P\{q_j}} = ⊥ then continue;
16         if M_P = ⊥ or f(g_P^Σ, l − l_P^min) < f(g_{P\{q_j}}^Σ + g, l − l_{P\{q_j}}^min) then
17           M_P = M_{P\{q_j}} ∪ {m};  // update best P-matchset at l to have m
18           g_P^Σ ← g_{P\{q_j}}^Σ + g; l_P^min ← l_{P\{q_j}}^min;
19     if M_Q ≠ ⊥ and (M = ⊥ or S < f(g_Q^Σ, l − l_Q^min)) then
20       M ← M_Q; S ← f(g_Q^Σ, l − l_Q^min);
21   return (M, S);
22 end
```

this case, a best P-matchset in $\mathcal{M}_2$ at $l^{(i)}$ can be found by adding $m^{(i)}$ to a best $(P \setminus \{q^{(i)}\})$-matchset at $l^{(i-1)}$. This claim can be proved by a simple "cut-and-paste" argument. Consider any matchset $M \in \mathcal{M}_2$. Since there are no matches within $(l^{(i-1)}, l^{(i)})$, $M \setminus \{m^{(i)}\}$ is a $(P \setminus \{q^{(i)}\})$-matchset at $l^{(i-1)}$. Hence, $s(M \setminus \{m^{(i)}\}, l^{(i-1)}) \leq s(M', l^{(i-1)})$, where M' is a best $(P \setminus \{q^{(i)}\})$-matchset at $l^{(i-1)}$. By the optimal substructure property of f,

$$\begin{aligned} & s(M \setminus \{m^{(i)}\}, l^{(i-1)}) \leq s(M', l^{(i-1)}) \\ \Rightarrow\ & s(M \setminus \{m^{(i)}\}, l^{(i)}) \leq s(M', l^{(i)}) \\ \Rightarrow\ & s(M, l^{(i)}) \leq s(M' \cup \{m^{(i)}\}, l^{(i)}). \end{aligned}$$

Therefore, $M' \cup \{m^{(i)}\}$ is a best P-matchset in $\mathcal{M}_2$ at $l^{(i)}$.

To summarize, then, we can compute $M_P^{(i)}$, a best P-matchset at $l^{(i)}$, by the following recurrence: $M_P^{(i)} =$

$$\begin{cases} M_P^{(i-1)}, \text{ if } q^{(i)} \notin P \text{ or } s(M_P^{(i-1)}, l^{(i)}) > s(M_{P\setminus\{q^{(i)}\}}^{(i-1)} \cup \{m^{(i)}\}, l^{(i)}); \\ M_{P\setminus\{q^{(i)}\}}^{(i-1)} \cup \{m^{(i)}\} \text{ otherwise.} \end{cases}$$

Algorithm 1 implements this recurrence with dynamic programming. It remembers, for every nonempty subset of query terms $P \subseteq Q$, a best P-matchset at the previous match location. From these matchsets, best P-matchsets at the current match location are calculated. The algorithm exploits the structure of the WIN scoring function to incrementally compute the scores.

Discussion The space complexity of Algorithm 1 is $O(|Q|2^{|Q|})$, because we must remember one best partial matchset for each subset of the query terms. The running time is $O(2^{|Q|} \sum_j |L_j|)$, because each match requires $O(2^{|Q|})$ time to compute the best partial matchsets. Although the complexity is still exponential in the number of query terms, the base of the exponent is small and constant. In contrast, the naive solution based on cross product is also exponential in $|Q|$, but has a much larger base, $|L_j|$, the number of matches.

IV. Distance-From-Median (MED) Scoring

As discussed in Section I, one problem with WIN is that window length alone cannot fully capture the degree of clusteredness in a matchset. The *distance-from-median (MED)* scoring function in this section addresses this problem. Intuitively, MED penalizes the score contribution of each individual match in the matchset by its distance from median location in the matchset. The longer the distance, the larger the penalty. In Figure 2, MED would score the second matchset higher because most of its matches are clustered around the median location. Formally, we define MED as follows.

Definition 5 (Distance-From-Median (MED) Scoring Function)**.** *Given a query Q and a matchset $M = \{m_1, \ldots, m_{|Q|}\}$, let* $\text{median}(M)$ *denote the median of M's match locations, i.e.,* $\text{median}(M) \stackrel{def}{=} \text{median}\{\text{loc}(m) \mid m \in M\}$.[2] *The* distance-from-median (MED) scoring function *has the following form:* $\text{score}_{MED}(M, Q) \stackrel{def}{=}$

$$f\Big(\sum_j \Big(g_j(\text{score}(m_j, q_j)) - |\text{loc}(m_j) - \text{median}(M)|\Big)\Big),$$

where f and g_j ($1 \leq j \leq |Q|$) are monotonically increasing functions. We call $c_j(m_j, l) \stackrel{def}{=} g_j(\text{score}(m_j, q_j)) - |\text{loc}(m_j) - l|$ the distance-decayed score contribution *(or* contribution *for short) of match m_j at location l.*

Again, we have intentionally kept the definition general by leaving functions f and g_j's unspecified. As a concrete example, consider the following scoring function:

$$\prod_j \big(\text{score}(m_j, q_j) \times e^{-\alpha|\text{loc}(m_j) - \text{median}(M)|}\big), \tag{3}$$

This scoring function multiplies together the individual match scores, and weighs each of them down by exponentially decaying it with rate $\alpha > 0$ over its distance to the median location. It can be seen as a natural extension of the WIN scoring function in (1) inspired by Cheng et al. [8]. It is not difficult to see that the above scoring function is a MED scoring function, with $f(x) = e^{\alpha x}$ and $g_j(x) = \ln(x)/\alpha$.

Overall Algorithm We present an algorithm that works for any MED scoring function, provided that f is monotonically increasing. The observation underpinning the algorithm is stated in the lemma below.

Definition 6 (Dominating Match)**.** *Given two matches m and m' for the same query term, we say that m* dominates *m' at location l if the (distance-decayed score) contribution of m is greater than or equal to that of m' at location l.*

A match m is dominating *at location l (for its query term) if, at l, m dominates all matches for its query term (i.e., m maximizes the contribution at l).*

Lemma 1. *Suppose that for a match m_j in a matchset M, there exists a match m'_j for the same query term q_j, such that m' dominates m at* $\text{median}(M)$*, i.e.,* $c_j(m', \text{median}(M)) \geq$

[2] We define the median of a multiset of size n to be the $\lfloor \frac{n+1}{2} \rfloor$-th ranked element when the elements are ranked by value, with the 1st ranked element having the greatest value.

$c_j(m, \text{median}(M))$. *Then the matchset* $M' = M \setminus \{m_j\} \cup \{m'_j\}$ *has the same or a higher* MED *score than* M.

Upon closer examination, the validity of this lemma is far from obvious. The criterion by which we replace m_j with m'_j is defined with respect to $\text{median}(M)$. However, this replacement may shift the median from $\text{median}(M)$ to $\text{median}(M')$, where the MED score of M' is defined. Therefore, it remains unclear whether the replacement could net a loss in MED score. A non-trivial proof of Lemma 1 is presented in the appendix.

By Lemma 1, we can always find an overall best matchset M, such that for each match $m_j \in M$, m_j is dominating at $\text{median}(M)$ for its query term. This observation leads to a simple and elegant solution for finding an overall best matchset. We examine each match (from all match lists) in turn in location order. Suppose we are examining a match m for query term q. We simply find, for each query term other than q, a dominating match at $\text{loc}(m)$. In case of ties (where multiple matches achieve the same maximum contribution), we always pick one that succeeds m in processing order, if such a match exists.[3] Then, we check if the matchset consisting of m and the $|Q|-1$ dominating matches indeed has its median located at m. If yes, we have found a candidate overall best matchset; if its score is higher than the highest we have encountered, we remember it as the overall best matchset found so far. Once we finish processing all matches, we will have found an overall best matchset.

Precomputation A naive implementation of the above algorithm would be quadratic in the size of the match lists, because at each match location we need to find dominating matches at this location for other query terms. Using a linear-time precomputation step, we can make it a constant-time operation to find a dominating match for a given query term at a given location. Intuitively, the precomputation step computes, for each match list L_j, a *dominating match function* U_j, which returns a dominating match in L_j at a given location.[4] We also define, for each match list L_j, the *contribution upper envelope* $S_j(l) \stackrel{\text{def}}{=} \max_{m \in L_j} c_j(m, l)$, i.e., the maximum contribution at l, which is achieved by $U_j(l)$. Figure 3 illustrates these two concepts. Given the simple shape of the contribution upper envelope S_j, we can record U_j simply by a list of dominating matches, one for each local maximum of S_j.[5]

The precomputation step works as follows. For each match list L_j, we process it sequentially while maintaining a stack of matches. To process a match m, we check whether m dominates the match at the top of the stack at $\text{loc}(m)$. If not, we discard m and move on. Otherwise, we pop from the stack any match m' that is dominated by m at $\text{loc}(m')$, until the stack is empty or we encounter an m' not dominated by m at $\text{loc}(m')$; we then push m onto the stack. After we finish processing L_j, the stack contains, from bottom to top, the list of matches representing a dominating match function U_j, ordered by location. Denote this list by V_j.

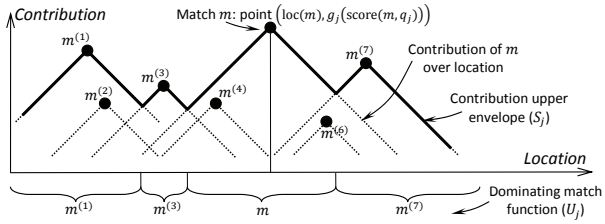

Fig. 3. Dominating match function U_j and contribution upper envelope S_j for match list L_j under MED. The contribution of a match m peaks at $\text{loc}(m)$ and drops off with a slope of -1 as we move away.

Algorithm 2: Computing overall best matchset for MED.

```
1  MaxJoinMED(Q, L_1, ..., L_|Q|) begin
2    M ← ⊥; S ← ⊥;      // M: overall best matchset found so far; S: its score
3    foreach query term q_j ∈ Q do
4      V_j ← PrecomputeDomMatchFunc(L_j, q_j); v_j^last ← ⊥;
5    foreach match m ∈ L_1 ∪ ··· ∪ L_|Q| in location order do
6      foreach query term q_j ∈ Q do          // advance the V_j's to loc(m)
7        while V_j ≠ ⊥ and head(V_j) ≤ loc(m) do
8          v_j^last ← head(V_j); V_j ← rest(V_j);
9      M_c ← {m};          // candidate matchset to be constructed around m
10     c_r ← 0;          // number of matches in M_c following m in location order
11     foreach query term q_j other than the one m matches do
12       m_1 ← v_j^last; m_2 ← head(V_j);
13       if m_2 ≠ ⊥ and (m_1 = ⊥ or dominates(m_2, m_1, q_j, loc(m))) then
14         M_c ← M_c ∪ {m_2}; c_r ← c_r + 1;
15       else
16         M_c ← M_c ∪ {m_1};
17     if c_r + 1 = ⌊(|Q|+1)/2⌋ then      // M_c is a candidate overall best matchset
18       if M = ⊥ or score_MED(M_c, Q) > S then
19         M ← M_c; S ← score_MED(M_c, Q);
20   return (M, S);
21 end
22 dominates(m, m', q_j, l) begin
23   return c_j(m, l) ≥ c_j(m', l);
24 end
25 PrecomputeDomMatchFunc(L_j, q_j) begin
26   S ← empty stack;
27   foreach match m ∈ L_j in order do
28     if ¬dominates(m, top(S), q_j, loc(m)) then continue;
29     while dominates(m, top(S), q_j, loc(top(S))) do pop(S);
30     push(S, m);
31   return S;
32 end
```

With the precomputed V_j's, the main algorithm is now able to find a dominating match for a given query term and a given location in constant time. Recall that the algorithm processes matches in location order, so it also issues requests for dominating matches in location order. Conveniently, matches in V_j's are ordered by location too, allowing us to service all requests for dominating matches by scanning V_j's in parallel with the match lists. The dominating match in V_j for a particular location can be found by comparing the contribution from up to two matches in V_j located closest to the given location (one to the left and one to the right). The detailed algorithm (including the precomputation step) is presented in Algorithm 2.

Discussion Because of the precomputed V_j's, the space complexity of Algorithm 2 is $O(\sum_j |L_j|)$, i.e., linear in

[3]To guarantee that the algorithm will find an overall best matchset, we need to consistently favor picking a dominating match that succeeds (or precedes) the current match, for every match considered. Interested readers may refer to our technical report [21] for details.

[4]Ties are broken by returning the dominating match that comes last in L_j.

[5]Strictly speaking, the list corresponds to the local maxima of S_j plus the tie-breaking dominating matches.

the size of the match lists. The precomputation step takes $O(\sum_j |L_j|)$ time, since each match at most can be pushed once and popped once. After precomputation, the algorithm takes $O(|Q| \sum_j |L_j|)$, because each match requires us to construct and check a matchset. Overall, the running time is $O(|Q| \sum_j |L_j|)$.

V. Maximize-Over-Location (MAX) Scoring

MED uses the median location in a matchset as a reference point to compute the distance-decayed score contributions of individual matches. Another natural choice for a reference point would be the location where the total contribution is maximized (which is often not the median location). The following definition formalizes this type of scoring functions.

Definition 7 (Maximize-Over-Location (MAX) Scoring Function)**.** *Given a query Q and a matchset $M = \{m_1, \ldots, m_{|Q|}\}$, the* maximize-over-location (MAX) scoring function *has the following form:* $\text{score}_{MAX}(M, Q) \stackrel{def}{=}$

$$\max_l f\Big(\sum_j g_j\big(\text{score}(m_j, q_j),\ |\text{loc}(m_j) - l|\big)\Big),$$

where f is a monotonically increasing function, and $g_j(x, y)$ ($1 \le j \le |Q|$) are monotonically increasing in x and monotonically decreasing in y. We call $c_j(m_j, l) \stackrel{def}{=} g_j\big(\text{score}(m_j, q_j), |\text{loc}(m_j) - l|\big)$ the (distance-decayed score) contribution *of match m_j at location l.*

While MED chooses the reference point based purely on the locations of matches, MAX bases this choice on both match locations and scores, by maximizing the matchset score over all possible reference point location l. Consequently, MAX tends to choose reference points near high-scoring matches. This choice captures the intuition that we want to "anchor" a matchset around matches we are most confident about.

We give two specific examples of a MAX scoring function. The first one is essentially a generalization of the MED scoring function in (3):

$$\max_l \prod_j \big(\text{score}(m_j, q_j) \times e^{-\alpha|\text{loc}(m_j) - l|}\big), \tag{4}$$

where $\alpha > 0$. Casting it in the terms of Definition 7, $f(x) = e^x$, and $g_j(x, y) = \ln(x) - \alpha y$.

The second example is a variation of the above, where we add (instead of multiply) the distance-weighted individual match scores together:

$$\max_l \sum_j \big(\text{score}(m_j, q_j) \times e^{-\alpha|\text{loc}(m_j) - l|}\big), \tag{5}$$

where $\alpha > 0$. In the terms of Definition 7, f is the identity function, and $g_j(x, y) = xe^{-\alpha y}$ for this scoring function. This function generalizes the scoring function of Chakrabarti et al. [7], which simply sets l to be the location of the match for the single "type" term in their query. We also use exponential decay to approximate their empirically measured distance-decay function.

In the remainder of this section, we first outline an approach for computing the overall best matchset that works for any f and g_j's. However, for a complex MAX scoring function, this approach has high computational complexity. Next, we give an efficient algorithm targeting MAX scoring functions satisfying certain properties. In fact, scoring functions in both (4) and (5) are amenable to the efficient algorithm.

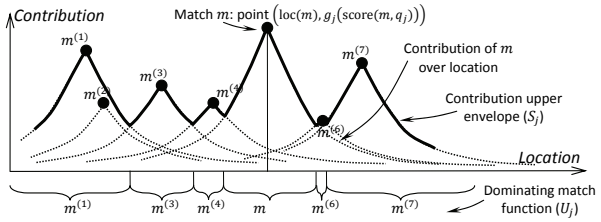

Fig. 4. Dominating match function U_j and contribution upper envelope S_j for match list L_j under the MAX scoring function (5). The contribution of a match m peaks at $\text{loc}(m)$ and drops off exponentially as we move away.

A General Approach Recall from Section IV the concepts of dominance (Definition 6), dominating match function (U_j), and contribution upper envelope (S_j). Their respective definitions are identical in this setting, except that we use the definition of contribution in Definition 7 instead of Definition 5. Specifically, $U_j(l) = \arg\max_{m \in L_j} g_j(\text{score}(m, q_j), |\text{loc}(m) - l|)$, and $S_j(l) = \max_{m \in L_j} g_j(\text{score}(m, q_j), |\text{loc}(m) - l|)$. Note that MAX's definition of contribution is more general than MED's. Figure 3 is still a good illustration of U_j and S_j for the MAX scoring function (4) (in the case of $\alpha = 1$). On the other hand, the MAX scoring function (5) have very differently shaped contribution upper envelopes, illustrated in Figure 4.

The approach is to first compute U_j and S_j for each query term q_j. Next, compute $l_{\text{MAX}} = \arg\max_l \sum_j S_j(l)$. Then, the matchset $\{U_1(l_{\text{MAX}}), \ldots, U_{|Q|}(l_{\text{MAX}})\}$ is an overall best matchset, as the lemma below shows (see [21] for proof).

Lemma 2. *$\{U_1(l_{MAX}), \ldots, U_{|Q|}(l_{MAX})\}$ is an overall best matchset under the MAX scoring function.*

Although conceptually simple, the above approach can be expensive for MAX scoring functions with complex g_j's. In particular, even though the contributions monotonically decrease with distance, the rate of decrease may still fluctuate, resulting in a complex U_j. The complexity of U_j can be measured by the number of interval-match pairs needed to represent it, where in each pair (I, m), I is maximal interval such that for every location $l \in I$, $U_j(I) = m$. If the contribution curves for different matches in a match list intersect each other many times, as illustrated in Figure 5, the number of interval-match pairs can be arbitrarily large (up to the number of all possible locations). Furthermore, the cost of computing $l_{\text{MAX}} = \arg\max_l \sum_j S_j(l)$ is linear in the total number of interval-match pairs for representing U_j's. Next, we describe a more efficient algorithm that specializes in MAX scoring functions with certain properties.

An Efficient Specialized Algorithm We consider two properties of the MAX scoring function that enable an efficient algorithm with complexity linear in the size of match lists.

Definition 8 (At-Most-One-Crossing and Maximized-At–Match)**.** *A contribution function c_j is* at-most-one-crossing *if for any two matches m and m' from a same match list L_j,*

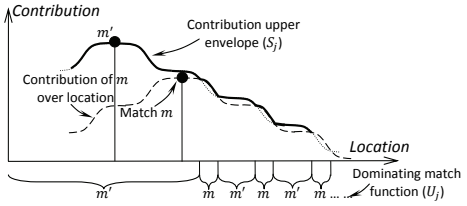

Fig. 5. A complex contribution upper envelope caused by two intersecting contribution curves, both monotonically decreasing over distance. For every two consecutive intersections, an interval-match pair is needed to present U_j.

the difference between their contributions, $c_j(m,l)-c_j(m',l)$, changes sign at most once over all possible location l. A MAX *scoring function is* at-most-one-crossing *its associated contribution functions are at-most-one-crossing.*

A MAX *scoring function is* maximized-at-match *if for any matchset $M = \{m_1, \ldots, m_{|Q|}\}$, there exists $m_k \in M$ such that* $\text{score}_{MAX}(M,Q) = f\big(\sum_j c_j(m_j, \text{loc}(m_k))\big)$.

Intuitively, the at-most-one-crossing property ensures the simplicity of dominating match functions—no two contribution curves can cross more than once. It is easy to see that the number of interval-match pairs needed to represent U_j is no more than $|L_j|$, the size of the corresponding match list.

The maximized-at-match property simplifies the job of computing l_{MAX}. With this property, the MED score of an overall best matchset is achieved at one of its match locations. Furthermore, every match in this matchset must be a dominating match at this location for the corresponding match list (otherwise, replacing that match with a dominating match would give a better matchset). Therefore, instead of solving a function minimization problem over the domain of all possible locations, we only need to look for l_{MAX} among the locations of dominating matches.

These properties are not selected arbitrarily; in fact, functions (4) and (5) both have these properties (see [21] for proof).

Lemma 3. *The* MAX *scoring functions in* (4) *and* (5) *satisfy both at-most-one-crossing and maximized-at-match properties.*

The algorithm starts with a precomputation step that computes and remembers the list of dominating matches V_j for each match list L_j, by sequentially processing L_j while maintaining a stack. This step is identical to the precomputation step of Algorithm 2, except that different contribution functions are used in testing dominance.

Next, the algorithm proceeds through the dominating matches in V_j's in location order. At each match location $\text{loc}(m)$, we consider the matchset consisting of a set of dominating matches (one from each V_j) for this location. Identification of the dominating match in V_j for a given location is again similar to Algorithm 2. If the matchset has a higher score than what we have previously encountered, we remember the matchset as the overall best found so far. An overall best matchset will be found once we finish processing all matches in V_j's.

Because of space constraints, the detailed algorithm is presented in [21]. It reuses many components of Algorithm 2 (with different contribution functions). Similar to Algorithm 2, the space complexity is $O(\sum_j |L_j|)$ (due to precomputation), and the overall running time is $O(|Q| \sum_j |L_j|)$.

VI. Avoiding Duplicate Matches

Thus far, to simplify discussion, we have not yet considered the possibility of duplicates across match lists. In many applications, however, such duplicates can arise. For example, consider the query {"asia," "porcelain"}. A single token "china" matches both query terms, and would appear as a match in match lists for both "asia" and "porcelain." In any given context, however, "china" can take on only one meaning and therefore should not match both simultaneously. A better matchset for the query would come from "fine ceramics from Jingdezhen," where "ceramics" is a match for "porcelain" and "Jingdezhen" is a match for "asia." However, our earlier problem formulations would have deemed {"china," "china"} a better matchset, because "china" matches "asia" better than "Jingdezhen" does, and, more importantly, the duplicate matches incur less distance-based penalty.

To avoid duplicate matches, we modify our definition of the overall-best-matchset problem as follows. We say that a matchset is *valid* if it contains no duplicate matches. We then restrict the matchsets considered in Definition 2 to valid matchsets only. The best-matchset-by-location problem can be similarly modified.

We present a simple and generic method that avoids duplicate matches for the overall-best-matchset problem. It is generic in that it works with any duplicate-unaware algorithm. The basic idea is to first run the duplicate-unaware algorithm to find an overall best matchset. If it happens to be duplicate-free, we are done. Otherwise, based on the duplicates in it, we create modified problem instances and rerun the duplicate-unaware algorithm over them.

The method is best illustrated by a simple example. Suppose the duplicate-unaware algorithm $\mathcal{A}$ returns a matchset in which matches m_1 and m_2 are duplicated: m_1 is used to match query terms q_{11} and q_{12}, while m_2 is used to match q_{21}, q_{22}, and q_{23}. We rerun $\mathcal{A}$ on $2 \times 3 = 6$ modified problem instances. Each instance is created from the original problem instance by removing m_1 and m_2 from match lists—specifically, m_1 from one of $\{L_{11}, L_{12}\}$, and m_2 from two of $\{L_{21}, L_{22}, L_{23}\}$. Each instance corresponds to a different way of ensuring that m_1 can match at most one of $\{q_{11}, q_{12}\}$ and that m_2 can match at most one of $\{q_{21}, q_{22}, q_{23}\}$. If for any modified instance, $\mathcal{A}$ still returns matchset with duplicates, the method is recursively applied to that instance. Finally, the method returns the best duplicate-free matchset found by $\mathcal{A}$ among all modified problem instances. Details are presented in [21].

The complexity of this method depends how many times it invokes $\mathcal{A}$. In the worst case, we may need to consider all modified instances in which some subset of the duplicates is removed. In practice, however, the method is very efficient on realistic inputs. Even if the input text contains many ambiguous tokens that could be duplicated in a matchset, we only need to run the duplicate-unaware algorithm once as long as the best matchset identified has no duplicates. Our method

takes full advantage of the common cases where duplicates are rare in best matchsets and simple to correct.

Algorithms with better worst-case bounds are possible, but are beyond the scope of this paper; we refer interested readers to our technical report [21] for additional details.

VII. Returning Best Matchset by Location

As motivated in Section I, for some applications it is not enough to find just one best matchset over the entire match lists. We now discuss a problem formulation that allows multiple matchsets to be returned. The intuition is that multiple desirable matchsets may occur throughout the input sequence, and each of them is "locally optimal." This intuition leads us to the following problem formulation, which returns best matchsets by their "anchor" locations.

Definition 9 (Anchor of a Matchset)**.** *The* anchor location *of a matchset $M = \{m_1, \ldots, m_{|Q|}\}$, denoted* $\mathrm{anchor}(M)$*, is defined as follows. For WIN, the anchor location is the largest match location in M, i.e.,* $\mathrm{anchor}(M) = \max_j \mathrm{loc}(m_j)$*. For MED, the anchor location is the median match location in M, i.e.,* $\mathrm{anchor}(M) = \mathrm{median}(M)$*. For MAX, the anchor location is the location where the score is maximized, i.e.,*

$$\mathrm{anchor}(M) = \arg\max_l f\Big(\sum_j g_j\big(\mathrm{score}(m_j, q_j), |loc(m_j) - l|\big)\Big)$$

(cf. Definition 7).

Definition 10 (Best-Matchset-by-Location Problem)**.** *Given query Q and associated match lists, the* best-matchset-by-location problem *finds, for each possible anchor location l, a best matchset M anchored at l; i.e., $M = \arg\max_{\mathrm{anchor}(M)=l} \mathrm{score}(M, Q)$.*

For the WIN scoring function, Algorithm 1 requires only minor modification to solve the above problem. Recall that Algorithm 1 processes matches in location order. When processing a match $m^{(i)}$ for query term $q^{(i)}$, we would identify the matchset consisting of $m^{(i)}$ and the best $(Q \setminus \{q^{(i)}\})$-matchset as a candidate matchset anchored at $\mathrm{loc}(m^{(i)})$. As soon as we finish processing all matches located at $\mathrm{loc}(m^{(i)})$, we can return the best candidate matchset located at $\mathrm{loc}(m^{(i)})$. The complexity of the algorithm remains $\mathrm{O}(2^{|Q|} \sum_j |L_j|)$.

For MED, we cannot directly extend Algorithm 2 to find best matchsets by location. While Lemma 1 ensures that an overall best matchset contains only dominating matches at its anchor, a subtlety is that a "locally best" matchset M with a specific anchor may in fact contain some non-dominating matches. Nonetheless, it can be shown that every match in M must dominate, at $\mathrm{anchor}(M)$, all other matches for the same query term located on the same side of $\mathrm{anchor}(M)$; there are up to $2|Q|-2$ such candidate matches (other than the anchor). Thus, after the precomputation step, when considering each match m, we switch to dynamic programming to choose $|Q|-1$ candidate matches—$\lfloor \frac{|Q|}{2} \rfloor$ to the left of $\mathrm{loc}(m)$ and the rest to the right—that, together with m, form the best matchset anchored at $\mathrm{loc}(m)$. The complexity of this algorithm is $\mathrm{O}(|Q|^2 \sum_j |L_j|)$. See [21] for details.

Finally, for MAX, we can solve the best-matchset-by-location problem by a simple modification to the algorithm in Section V. After precomputation, instead of going through only the dominating matches in V_j's, we go through all match locations in the match lists, and compute, for each location l, the best matchset anchored at l, which consists of dominating matches at l. The V_j's are still used to identify dominating matches. The complexity remains $\mathrm{O}(|Q| \sum_j |L_j|)$.

A Note on Streaming A related issue is whether we can develop *streaming* algorithms for the best-matchset-by-location problem. A streaming algorithm would make a single pass over all match lists in parallel, and return a best matchset for an anchor location once it has been identified.

For WIN scoring functions, Algorithm 1, extended for the best-matchset-by-location problem as described above, is streaming. A result matchset is returned as soon as its last match is processed. The space required by the algorithm is independent of the size of the input match lists.

For MED and MAX, however, the problem is fundamentally not amenable to good streaming solutions. The reason is that in a matchset, the anchor location comes before the last match location, and the two can be arbitrarily far apart. In general, we cannot return any result matchset until we have seen the end of a match list, because the very last match in the list, no matter how far from the anchor, may have an individual match score just high enough to make this match part of the best matchset at the anchor.[6] In practice, however, incompatibility of MED and MAX with streaming is not an issue for our applications. The input match lists (e.g., derived from a document) are finite, so even if they can be accessed only once, we can cache them for later access. By further exploiting properties of the scoring function and assuming upper bounds on individual match scores (e.g., if all of them are in $(0, 1]$), it should be possible to develop less blocking algorithms that prune their state more aggressively and return result matchsets earlier; they are an interesting direction for future work.

VIII. Experiments

We implemented all proposed algorithms (with the duplicate-handling method in Section VI) in C++. We also implemented three naive algorithms NWIN, NMED, and NMAX, which exhaustively generate all possible matchsets and pick the one with the highest score for WIN, MED, and MAX scoring functions, respectively. Analytically, their time complexities are $\Theta(|Q| \prod_{j=1}^{|Q|} |L_j|)$ (Section II).

We conduct all our experiments on a single-core 3.6GHz desktop computer running CentOS 5 with 1GB memory. We measure the wall-clock time of execution when the computer is otherwise unloaded. We exclude the time to generate input match lists, since it is common to all algorithms. The execution times are quite consistent. We repeated the experiments

[6] For this very reason, modified MED and MAX algorithms for the best-matchset-by-location problem, as described earlier, still make two passes over the input match lists.

10 times for a large number of data points and found the coefficient of variation to be only 5.7% on average.[7]

We evaluate our algorithms on three datasets: One is synthetic; the other two are from TREC (`trec.nist.gov`) and DBWorld (`www.cs.wisc.edu/dbworld`).

Synthetic Dataset We use a synthetic dataset generator to control various factors influencing the performance of our algorithms. In particular, we consider: number of query terms, total size of the match lists in a document, frequency of duplicates (cf. Section VI),[8] and skewness in the sizes of match lists (or the relative popularities of query terms).

Given a match location, the generator determines τ, the number of matches (across match lists) at this location, according to an exponential distribution with density $p(\tau) \propto \lambda e^{-\lambda\tau}$ over the range of τ between 1 and the number of query terms. Larger λ means a higher probability of picking a smaller τ, and therefore a lower frequency of duplicates.

The skewness in the sizes of the match lists are controlled by a Zipf distribution $f(k;s) \propto \frac{1}{k^s}$, which states that the popularity of a query term is inversely proportional to its popularity rank k (with the most popular one ranked first) raised to power s. Increasing s leads to more skewness in sizes of the match lists.

The experiments below are run on synthetic datasets each consisting of 500 documents, with an average of 1000 words each. By default, the number of query terms is set to be 4; the total size of the match lists is set at 30 per document; parameter λ is set to 2.0 (which translates to a little less than 24% duplicates); parameter s is set to 1.1. The locations of matches are chosen at random. Individual match scores are drawn uniformly randomly from $(0, 1]$. All documents are relevant to the query and each algorithm is run on every document. For each algorithm, we report the total execution time over the entire set of 500 documents.

First, we vary the number of terms in a query from 2 to 7 (Figure 6). Note the performance gain by the proposed algorithms over their naive counterparts. The combinatorial explosion of possible matchsets in the naive algorithms' search spaces lends an argument to this difference. NMED fares worse than NWIN because of median calculation. NMAX is even slower, because it does not know the anchor of a matchset a priori (any match location in the matchset can potentially maximize the total contribution); hence, it needs to compute the total contribution at every match location in the matchset. Among the three proposed algorithms, WIN fares worse than MED and MAX because of an additional $2^{|Q|}$ term in its running time. However, this difference is not huge, because the number of query terms in practice are not large enough to induce a significant difference.

[7]Only 4 out of the 36 data points we measured had a coefficient of variation greater than 10%, and the worst was no more than 27.3%, not significant enough to affect the conclusions of our experiments.

[8]A match in some match list is counted as a duplicate if its location is identical to at least one match from another match list. We define the frequency of duplicates to be the number of duplicates divided by the total size of the match lists.

Next, we vary the total size of the match lists per documents from 10 to 40 (Figure 7). As the number of matches increases, there is an exponential growth in the execution times of the naive algorithms. In contrast, our proposed algorithms hold steadily close to the horizontal axis. It does not take very long match lists to realize significant performance advantages.

In the third experiment, we vary λ in the exponential distribution from 1.0 to 3.0 (Figures 8 and 9); accordingly, the frequency of duplicates changes roughly from 60% to 10%. This decrease causes fewer repetitions of our duplicate-unaware algorithms by our duplicate-handling method (cf. Section VI). In Figure 8, we see that even with an unrealistically high frequency of duplicates (60%), the duplicate-unaware algorithms repeat only between 10 to 12 times on an average. Since each repetition is efficient, the few number of repetitions imply that the total execution times of our approaches remain significantly better than naive ones even with a lot of duplicates, as Figure 9 shows.

Finally, we vary s in the Zipf distribution controlling the skewness in the size of posting lists (Figure 10). As skewness increases, the number of possible matchsets, which is the product of the sizes of the match lists, decreases. Therefore, performances of the naive algorithms improve. However, they remain worse than our algorithms, catching up only when $s = 4$. With such extreme skewness, all match lists are of size 1 except one match list.

TREC 2006 QA Dataset Our second dataset comes from the question answering task of TREC 2006 [1]. This task specifies a set of questions which need to be answered from a collection of documents. Since the focus of our paper is on algorithmic efficiency, the primary goal of this experiment is to compare and understand the execution times of various algorithms on a real dataset. Previous work, such as [7, 8], has already demonstrated the effectiveness of the approach when combined with methods of producing high-quality match lists and individual scores. For this experiment, we implemented a simple matcher to identify and score individual matches. Two terms are considered to be matching if their WordNet (`wordnet.princeton.edu`) graph distance d (in number of edges) is no more than 3; we score this match by $(1 - 0.3d)$.[9] We use the stem of a word as returned by a standard Porter's stemmer in all our string comparisons.

For this experiment, we consider only *factoid queries* in the TREC task, which expect a fact (as opposed to a list of facts) as an answer, e.g., *Where was Shakespeare born?* All factoid queries in this TREC task can be converted to multi-term queries. Because of our limited WordNet-based matcher, we do not consider queries whose query or answer terms do not appear in WordNet. For example, *Coriolanus* is not in WordNet, so we have no hope of identifying it as a play (let alone Shakespeare's). The seven queries we selected, shown in Figure 12, form a representative sample of queries that can be

[9]The detailed scoring functions we used for this experiment are as follows. For WIN, $g_j(x) = x/0.3$ and $f(x, y) = x - y$. For MED, $g_j(x) = x/0.3$ and $f(x) = x$. For MAX, we use Eq. (5) with $\alpha = 0.1$.

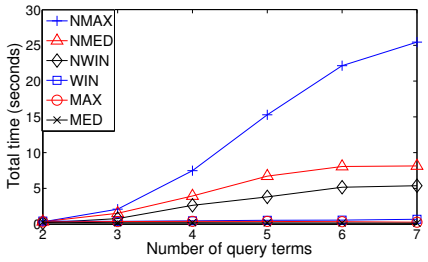

Fig. 6. Execution times when increasing the number of terms in a query.

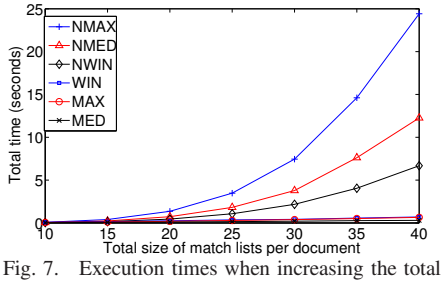

Fig. 7. Execution times when increasing the total size of match lists per document.

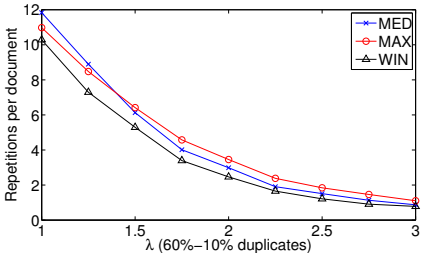

Fig. 8. Number of times a duplicate-unaware algorithm is executed per document when decreasing the frequency of duplicates.

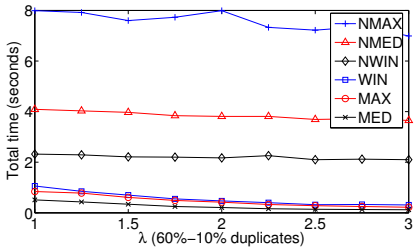

Fig. 9. Execution times when decreasing the frequency of duplicates.

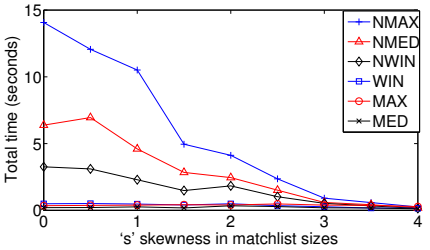

Fig. 10. Execution times when increasing the skewness in the popularities of query terms.

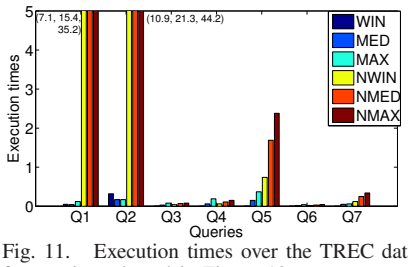

Fig. 11. Execution times over the TREC dataset for queries selected in Figure 12.

handled by WordNet. For each query q, we run all algorithms over the 1000 short documents (averaging 450-500 words per document) associated with q by TREC.[10] The fourth column of the table in Figure 12 shows the average sizes of q's match lists in a document, and the fifth column shows the average number of duplicate matches per document.

The running times (over all 1000 documents) are shown in Figure 11 for each query and each algorithm. Note the bars corresponding to naive algorithms for Q1 and Q2 are truncated because they are off-scale (one to two orders of magnitudes worse than our algorithms). Also, note that for queries with three terms or less, the scoring functions WIN and MED are actually identical; in these cases, we simply invoke MED instead of WIN. Therefore, bars for WIN are omitted for Q3–Q7 in Figure 11.

Whenever there are several match lists of moderate sizes (Q1, Q2), or one good-size match list with enough "support" from other lists (Q5 and to a lesser extent Q7), we see significant performance advantages of our algorithms. For Q3, Q4, and Q6, the naive algorithms perform well, because there is a large skew in the popularities of terms in these queries (cf. Figure 12), dramatically decreasing the total number of possible matchsets. This issue can be addressed by a simple fix to our algorithms: If all match lists but one contain no more than one match each, we switch to a naive algorithm. In any case, the saving is small compared with the differences in performance of harder queries.

As discussed earlier, we do not intend this experiment to evaluate the quality of information retrieval; nonetheless, we make some observations here. For each scoring function, we rank the documents by their overall best matchset scores. The fifth column of the table in Figure 12 shows, for each of the scoring function, the "answer rank," which is the rank of a document in which the best matchset found is the correct answer. Number of documents tied for this rank are indicated in brackets. There was only one document with rank 1 for all the queries, except for WIN's execution on $Q2$. From the table, we see very reasonable results despite our simple matcher.

DBWorld CFPs For this experiment, we collected messages posted through the DBWorld mailing list during June 24-26, 2008. Out of the total of 38 messages, 25 were emails announcing conferences, workshops, or other such meeting events. We execute the query {*conference*|*workshop*, *date*, *place*} on these 25 documents. By finding the overall best matchset in each document,[11] we hope to extract the date and the location of the meeting being announced.

The table below summarizes the results of this experiment:

avg. match list sizes per doc			# dups	avg. running time (ms) per doc				
conference|*workshop*	*date*	*place*	per doc	WIN	MAX	NWIN	NMED	NMAX
13.2	12.7	73.5	0	0.8	3.2	27.2	61.2	95.2

Note that the performance of MED is not shown because we can use WIN instead for a query with only three terms (as in the case of Q3–Q7 in the TREC experiment). The significant performance advantages of our algorithms over the naive ones can be explained by the large average match list sizes. It turns

[10] Queries in the QA task are divided into groups based on topic. For each topic, the QA task organizers provided a set of 1000 articles selected based on the questions in the topic (see [1] for details).

[11] Our matcher for *conference*|*workshop* is based on WordNet, which allows us to match synonyms such as *symposium*. We added an edge between *conference* and *workshop* in WordNet. The term *conference* itself is scored 1, while any term directly connected to *conference* in WordNet is scored 0.7. For *date*, we use a simple matcher that looks for month names and numbers between 1990 and 2010; identified matches are scored 1. For *place*, if a term can be found in the GeoWorldMap database (`www.geobytes.com`), we consider it a match with score 1. If GeoWorldMap does not have the term, we check if the term is directly connected to *place* in WordNet; if yes, it is considered a match with score 0.7. We added an edge between *university* and *place* in WordNet to improve accuracy. The three scoring functions are exactly the same as those used in the TREC experiment.

					answer rank		
ID	factoid query	query	match list sizes	# dups	MED	MAX	WIN
Q1	*Leaning Tower of Pisa began to be built in what year?*	*Leaning Tower of Pisa, began, build, year*	(2.9, 0.2, 8.3, 3.7)	0.6	1	1	1
Q2	*What school and in what year did Hugo Chavez graduate from?*	*Chavez, graduate, school, year*	(6.7, 5.2, 4.3, 4.6)	2.7	2(3)	1	1(2)
Q3	*In what city is the lebanese parliament located?*	*Lebanese Parliament, in, city*	(0.1, 11.9, 4.1)	0	1	1	1
Q4	*In what country was Stonehenge built?*	*country, Stonehenge, in*	(11.4, 0.04, 11.5)	0.8	1	1	1
Q5	*When did Prince Edward marry?*	*Prince Edward, marry, date*	(3.4, 2.1, 18.2)	0.7	1	1	1
Q6	*Where was Alfred Hitchcock born?*	*Alfred Hitchcock, born, city*	(3.6, 0.1, 8.4)	0	2(2)	2(2)	2(2)
Q7	*Where is the IMF headquartered?*	*IMF, headquarters, city*	(7.5, 1.0, 2.4)	0.4	1	1	1

Fig. 12. Selected queries from the TREC QA dataset.

out that CFPs contain a huge number of places because they often list PC members' affiliations. CFPs contain many dates as well, e.g., abstract submission and camera-ready deadlines.

Even with our simple matchers, we achieve reasonable accuracy. For 18 out of the 25 messages, all three scoring functions correctly identify the matchset containing the desired information. In 6 out of the remaining 7 messages, WIN is able to obtain a correct partial (two-term) matchset for the message. Meanwhile, MED and MAX are able to do so in 5 out of the remaining 7 messages.[12]

IX. Related Work

Many IR researchers [11, 9, 19, 18, 5, 20] have argued that integrating proximity into document scoring functions helps improving retrieval effectiveness. They are mostly concerned with scoring a document, whereas as we score matchsets. Nevertheless, many parallels can be drawn in the choices of scoring functions. In [11] and [9], the shortest interval containing a matchset is used as a measure of proximity, analogous to our WIN. In [18], an influence function assigns each position within the document a value that decreases with the distance to the nearest occurrence of a query term; documents are scored by combining the influence function of each query term, analogous to our MAX. Among other works, [19] and [5] measure pair-wise proximity in neighboring matches while [20] first groups nearby matches into a span and then measures the contribution of these spans.

More closely related to our work are systems for semantic search, information extraction, question answering, and entity search, e.g., [6, 14, 7, 8]. The Binding Engine [6] supports queries involving concepts, but depends heavily on matches being very close to each other. Avatar [14] maps rule-based annotators into database queries, and relies on the underlying database engine to process proximity-aware rules. We have discussed [7, 8] throughout the paper. Different from our work, these papers do not focus on developing new, efficient algorithms for processing match lists.

With the rise of semi-structured data and advances in information extraction, integration of database and IR techniques has attracted much attention [22]. One line of research in this direction is ranked keyword search over data with structures [3, 4, 15, 12, 16, 10, 17]. Some elements of the scoring functions, such as how to decay scores over distance and how to combine individual match scores, resemble ours. However, our problem has inherently lower complexity, because our matches are located on a line instead of a graph. Therefore, we are able to develop much more efficient algorithms.

The best-overall-matchset problem can be regarded as a multi-way join followed by $\max$-aggregation. There has been a lot of work on rank-aware query processing [13], including top-k joins. While the top-k join problem in its most general form subsumes ours, the assumptions commonly made by works in this area do not hold in our setting, so the solutions are not compatible. Specifically, they assume that the input lists are sorted by scoring attributes, and that the join scoring function is monotone in all its inputs. In our setting, however, the input to a matchset scoring function includes both scores and locations of individual matches. While the match lists are sorted by location, and our scoring function is monotone with respect to the proximity among locations, it is not monotone with respect to locations themselves.

Finally, the best-matchset-by-location problem may look similar to a join-aggregation problem in stream processing [2]. However, our problem is fundamentally different, because of the absence of a window in the stream processing sense and, in the case of MED and MAX the possibility of a "later" match contributing to the "current" answer (Section VII). Even for WIN, where we do have a streaming solution, the problem is not a stream processing one because the window in our scoring function is used to score matchsets, instead of restricting what matches can join. One could conceivably solve the problem by stream processing using a large enough window (and assuming some upper bounds on individual scores), but the solution still will not be as efficient as ours.

X. Conclusion

Inspired by applications in information retrieval and extraction, we have introduced the problem of weighted proximity best-joins, where input items have weight and location attributes, and the results are ranked by a scoring function that combines individual weights and the proximity among joining locations. We have considered three types of scoring functions that cover a number of variations used in applications. By exploiting properties of these scoring functions, we develop fast algorithms with time complexities linear in the size of their inputs. The efficiency of our algorithms, in both theory and practice, make them effective tools in scaling up information retrieval and extraction systems with sophisticated criteria for ranking answers extracted from documents.

[12]One might wonder if a heuristic that simply returns the first date in a document would work in this setting, without involving proximity. Unfortunately, it turns out that this heuristic works poorly, because the first date can often be something else, e.g., a new submission deadline in a deadline extension announcement (in fact, out of the 25 messages, 7 fall into this case, and our algorithms still manage to find correct answers for 6 of them).

REFERENCES

[1] TREC 2006 QA data, 2006. trec.nist.gov/data/qa/t2006_qadata.html.

[2] Aggarwal, editor. *Data Streams: Models and Algorithms*. Springer, 1st edition, Nov. 2006.

[3] Amer-Yahia, Botev, and Shanmugasundaram. TeXQuery: A full-text search extension to XQuery. In *WWW*, 2004.

[4] Balmin, Hristidis, and Papakonstantinou. Authority-based keyword queries in databases using ObjectRank. In *VLDB*, 2004.

[5] Büttcher, Clarke, and Lushman. Term proximity scoring for ad-hoc retrieval on very large text collections. In *SIGIR*, 2006.

[6] Cafarella and Etzioni. A search engine for natural language applications. In *WWW*, 2005.

[7] Chakrabarti, Puniyani, and Das. Optimizing scoring functions and indexes for proximity search in type-annotated corpora. In *WWW*, 2006.

[8] Cheng, Yan, and Chang. EntityRank: Searching entities directly and holistically. In *VLDB*, 2007.

[9] Clarke, Cormack, and Tudhope. Relevance ranking for one to three term queries. *Information Processing and Management*, 36(2), 2000.

[10] Gou and Chirkova. Efficient algorithms for exact ranked twig-pattern matching over graphs. In *SIGMOD*, 2008.

[11] Hawking and Thistlewaite. Proximity operators—so near and yet so far. In *TREC*, 1995.

[12] He, Wang, Yang, and Yu. BLINKS: Ranked keyword searches on graphs. In *SIGMOD*, 2007.

[13] Ilyas and Aref. Rank-aware query processing and optimization (tutorial). In *ICDE*, 2005.

[14] Jayram, Krishnamurthy, Raghavan, Vaithyanathan, and Zhu. *IEEE Data Eng. Bull.*, 29(1), 2006.

[15] Kacholia, Pandit, Chakrabarti, Sudarshan, Desai, and Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, 2005.

[16] Kasneci, Suchanek, Ifrim, Ramanath, and Weikum. NAGA: Searching and ranking knowledge. In *ICDE*, 2008.

[17] Li, Ooi, Feng, Wang, and Zhou. EASE: An effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *SIGMOD*, 2008.

[18] Mercier and Beigbeder. Fuzzy proximity ranking with boolean queries. In *TREC*, 2005.

[19] Rasolofo and Savoy. Term proximity scoring for keyword-based retrieval systems. In *ECIR*, 2003.

[20] Song, Taylor, Wen, Hon, and Yu. Viewing term proximity from a different perspective. In *ECIR*, 2008.

[21] Thonangi, He, Doan, Wang, and Yang. Weighted proximity best-joins for information retrieval. Technical report, Duke University, June 2008. www.cs.duke.edu/dbgroup/papers/2008-thdwy-multikw.pdf.

[22] Weikum. DB&IR: Both sides now (keynote). In *SIGMOD*, 2007.

APPENDIX

Proof of Lemma 1. Let $l = \mathrm{loc}(m_j)$, $l' = \mathrm{loc}(m'_j)$, $g = \mathrm{score}(m_j, q_j)$, and $g' = \mathrm{score}(m'_j, q_j)$. It is given that

$$\Delta = (g' - \left|l' - \mathrm{median}(M)\right|) - (g - |l - \mathrm{median}(M)|) \geq 0.$$

Our goal is to show $\mathrm{score}_{\mathsf{MED}}(M', Q) \geq \mathrm{score}_{\mathsf{MED}}(M, Q)$. Since f is monotonically increasing, an equivalent goal is to show $\Delta_1 + \Delta_2 \geq 0$, where

$$\Delta_1 = (g' - \left|l' - \mathrm{median}(M')\right|) - (g - |l - \mathrm{median}(M)|);$$

$$\Delta_2 = \sum_{i \neq j} |\mathrm{loc}(m_i) - \mathrm{median}(M)| - \sum_{i \neq j} \left|\mathrm{loc}(m_i) - \mathrm{median}(M')\right|.$$

Note that if $\mathrm{median}(M') = \mathrm{median}(M)$, then $\Delta_1 + \Delta_2 \geq 0$ obviously holds, because $\Delta_1 \geq 0$ (which follows from $\Delta \geq 0$) and $\Delta_2 = 0$.

Without loss of generality, let us assume that $\delta = \mathrm{median}(M') - \mathrm{median}(M) > 0$ (the case where $\delta < 0$ is symmetric). First, to derive Δ_2, consider the matches $M \cap M'$. We show that they can be partitioned into two disjoint sets:

- L, matches in $M \cap M'$ that are ranked (by location) at or below the median rank ($\lfloor \frac{|Q|+1}{2} \rfloor$) in M; and
- R, matches in $M \cap M'$ that are ranked (by location) at or above the median rank ($\lfloor \frac{|Q|+1}{2} \rfloor$) in M'.

We note that L and R are disjoint, because all match locations in L are no greater than $\mathrm{median}(M)$, while all match locations in R are no less than $\mathrm{median}(M')$. We also note that $L \cup R = M \cap M'$; i.e., $M \cap M'$ contains no match that is both ranked above the median in M and ranked below the median in M'. If there is such a match, its rank would have changed by at least 2. However, for any match in $M \cap M'$, its rank can only change by at most 1, because the removal of m_j can only move the match up by at most 1 while the addition of m'_j can only move the match down by at most 1.

For matches in L, their distance to $\mathrm{median}(M)$ is δ less than their distance to $\mathrm{median}(M')$; for matches in R, their distance to $\mathrm{median}(M)$ is δ more than their distance to $\mathrm{median}(M')$. Hence, $\Delta_2 = \delta(|R| - |L|)$.

Now, what are the sizes of L and R? We claim that $|R| - |L| \geq -1$. The reason is as follows. First, consider L. There are a total of $|Q| - \lfloor \frac{|Q|+1}{2} \rfloor + 1$ matches in M ranked at or below the median of M. One of them must be m_j; otherwise, it is impossible for median to move right with the removal of m_j and the addition of any m'_j. Hence, $|L| = |Q| - \lfloor \frac{|Q|+1}{2} \rfloor$. Next, consider R. There are a total of $\lfloor \frac{|Q|+1}{2} \rfloor$ matches in M' ranked at or above the median of M'. One of them must be m'_j; otherwise, it is impossible for median to move right with the removal of any m_j and the addition of m'_j. Hence, $|R| = \lfloor \frac{|Q|+1}{2} \rfloor - 1$, and $|R| - |L| = 2\lfloor \frac{|Q|+1}{2} \rfloor - |Q| - 1 \geq -1$. Therefore, $\Delta_2 = \delta(|R| - |L|) \geq -\delta$.

Finally, let us derive Δ_1. As argued above, m'_j must be ranked at or above the median of M', so $|l' - \mathrm{median}(M)| - |l' - \mathrm{median}(M')| = \delta$. Therefore, $\Delta_1 = \Delta + |l' - \mathrm{median}(M)| - |l' - \mathrm{median}(M')| = \Delta + \delta \geq \delta$. Combining with the fact that $\Delta_2 \geq -\delta$, we have $\Delta_1 + \Delta_2 \geq 0$. □