COMP6714	Lecture 2

Information Retrieval and Search Engines

Lecturer: Wei Wang *Date:*

1 Preliminaries

Before studying the MaxScore algorithm, it is beneficial to review and have a deeper understanding of the vanilla DAAD query processing algorithm. In particular, find out the similarities and differences between the DAAD algorithm and the algorithm to process the disjunctive Boolean query (i.e., if the query is `A B C`, the disjunctive query is `A or B or C`).

The DAAD algorithm, at a high-level, does two things:

1. **Candidate generation**: it gets a set of *candidate* documents, which is technically the union of all query keywords' inverted lists, and
2. **Soring**: for each candidate, it computes its score.

To further improve the algorithm, we need to reduce the potentially huge amount of candidates generated by the DAAD algorithm. Consider the following example: the query is `A B C`, and we denote the documents in `A`'s inverted list as C_A. The candidates generated by the DAAD algorithm is $\mathcal{S}_1 := C_A \cup C_B \cup C_C$. Now, can we reduce it to, say, $\mathcal{S}_2 := C_A \cup C_B$?[1]

Notice that the only set of candidates we will miss by using $\mathcal{S}_2$ instead of $\mathcal{S}_1$ is $C_C \setminus (C_A \cup C_B)$, or in other words, those documents that **only** contain `C`. What is the maximum possible score for these documents? It is *at most* $idf(\texttt{C}) \cdot \max_{d \in \mathcal{C}_2}\{tf(d, \texttt{C})\}$. If even this score is no larger than the currently found k-th highest score, then we can *safely* use $\mathcal{S}_2$ instead of $\mathcal{S}_1$.

2 MaxScore

2.1 Description

For every term t, we can quickly compute the *maxscore* from precomputed information stored in its postings list L. E.g., in VSM, it is just $idf(t) \cdot \max_{d_i \in L}\{tf(d_i, t)\}$.[2]

Without loss of generality, we assume that we have already rearranged the query terms in the decreasing order of their maxscores.[3]

The basic idea of the maxscore algorithm is based on the following observations:

- If we know a document does not appear in a set of terms' posting lists, its maximum *possible* score is the sum of the maxscores of the rest of the terms.
- Let τ' be the minimum score of the currently top-k scoring documents, we do not need to process a document whose maximum possible score is no larger than τ'.

[1] It is important to note that we still allow the algorithm to access the inverted list of `C` in the *scoring* phrase.
[2] Find out which part is precomputed.
[3] Think why we make such an assumption?

Hence, the maxscore algorithm optimizes the basic DAAT algorithm by trying to gradually remove the last query term (possibly repeatedly) from the so-called *required term set*.[4] Only documents in the postings lists of required term set are used to *drive* the DAAT algorithm. Other postings lists are only used to *score* a document (via skipping).

We show the pseudo code of the algorithm in Algorithms 1 to 4.

Algorithm 1: MaxScore($\{L_1, L_2, \ldots, L_m\}, k$)

Description: The main difference from the standard DAAT algorithm is that the algorithm is driven only by postings in *RTLists*.

Data: Postings lists L_is are ordered by their *maxscore* in *decreasing* order

Initialize min-heaps H and *topk*; `/* weights are` *docID* `and` *score*`, respectively. We push` k `negative values into` *topk* `initially. */`;

$RTLists \leftarrow \{L_1, L_2, \ldots, L_m\}$;

$PTLists \leftarrow \emptyset$;

forall L_i **do**

  H.push(L_i.curPosting(), L_i.curPosting().$docID$); L_i.next(); `/* curent posting contains all the necessary info for scoring plus the list id. */`;

while H.isEmpty() $\neq$ **true do**

  `/* score another doc */`

  $(score, docID) \leftarrow$ calcScore($H, PTLists$);

  `/* update the top-k heap and the top-k bottom score */`

  $topk.push(docID, score)$;

  $topk.pop()$;

  $\tau' \leftarrow topk$.peep().$score$;

  `/* update RTLists and PTLists based on` τ' `*/`

  update($\tau', RTLists, PTLists, H$);

Algorithm 2: calcScore($H, PTLists$)

Description: Collect all postings of *docID* in H into *info*, and call calcScore2() to compute the final score for *docID*.

Initialize *info*;

$(info, docID) \leftarrow H$.pop();

$i \leftarrow info.listID$;

H.push(L_i.curPosting(), L_i.curPosting().$docID$); L_i.next();

while H.peep().$docID = docID$ **do**

  $(tempInfo, docID) \leftarrow H$.pop();

  $info$.add($tempInfo$);

  $i \leftarrow tempInfo.listID$;

  H.push(L_i.curPosting(), L_i.curPosting().$docID$); L_i.next();

$score \leftarrow$ calcScore2($docID, info, PTLists$);

Note that we just illustrate a naive version of the pseudo-code which illustrates the main ideas. In actual implementation, there are many optimizations that must be added. For example, one can maintain the last τ' value and avoid calling the update function if τ' didn't change. The partitioning of all m lists into *RTLists* and *PTLists* can be done *incrementally*.

Algorithm 3: calcScore2($docID, info, PTLists$)

Description: Collect possible postings of $docID$ by seeking on $PTLists$, and then compute the final score.

forall $L_i \in PTLists$ **do**
  $id \leftarrow L_i$.skipTo($docID$);
  if $id = docID$ **then**
    $info$.add(L_i.curPosting());

$s \leftarrow$ compute score of $docID$ based on $info$;

Algorithm 4: update($\tau', RTLists, PTLists, H$)

Description: Update $RTLists$ and $PTLists$, and also remove items in H if it belongs to lists being moved to $PTLists$.

$upperBound \leftarrow 0$;
for $i = m$ **to** 1 **do**
  $upperBound \leftarrow upperBound + L_i.maxscore$;
  if $upperBound \geq \tau'$ **then**
    break;

$RTLists \leftarrow \{L_1, \ldots, L_i\}$;
$PTLists \leftarrow \{L_{i+1}, \ldots, L_m\}$;
Remove items in H that came from a list now in $PTLists$;

2.2 A Running Example

Consider the example in Table 1. We make many simplifying assumptions, including that the score contribution of a term is just its tf, and the final score of a document is the sum of scores from all the query terms it contains. We highlight the major event in each iteration of the algorithm below.

1. Initially, $RTList$ is all the three lists.
2. 1st iteration: H gives D_1, and we collect all its postings from H, and calculate its score as $2 + 1 = 3$. Hence, $\tau' = -1$, and there is no need to update $RTList$.
3. 2nd iteration: H gives D_2, and we collect all its postings from H, and calculate its score as $8 + 1 = 9$. Hence, $\tau' = 3$. We shrink $RTList$ to $\{A, B\}$.
4. 3rd iteration: H gives D_4 (**not** D_3), and we collect its postings from H, as well as finding its postings in $PTLists$, and calculate its score as $2 + 4 + 1 = 7$. Hence, $\tau' = 7$. We shrink $RTList$ to $\{A\}$.
5. 4th iteration: H becomes empty and we stop, and the final top-2 results are: $(D_2, 9)$ and $(D_4, 7)$.

2.3 Cost Analysis

The worst case complexity of the algorithm is the same as without the maxscore optimization. However, as we can see from the running example, if we are able to obtain k documents with high scores (they do not necessarily need to be the final results), the algorithm can be very efficient by scoring fewer number of documents and making use of the skipping capability of postings lists.

[Strohman et al., 2005] reports that a basic maxscore algorithm improves query time of a baseline DAAT algorithm by 40% and it scores only about 50% of the documents.

[4] We denote their postings lists as $RTLists$ in the algorithms.

term	maxscore	postings
A	8	$(D_1:2),(D_2:8),(D_4:2)$
B	4	$(D_1:1),(D_4:4),(D_{10}:1),(D_{11}:4),\ldots$
C	2	$(D_2:1),(D_3:2),(D_4:1),(D_{10}:2),(D_{11}:2),\ldots$

Table 1: A Running Example ($k = 2$ and Each Posting only Contains ($docID : tf$))

2.4 Bibliography Notes

We quote from [Shan et al., 2012]:

> The original description of the MaxScore [Turtle and Flood, 1995] strategy does not contain enough details, and it is different from a later implementation by Strohman [Strohman et al., 2005]. Jonassen and Bratsberg [Jonassen and Bratsberg, 2011] presented a more detailed MaxScore algorithm which combines the advantage of both Strohman's and Turtle's implementations.

Our description of the maxscore algorithm is a simplified version without much optimization.

Another way to make use of the per list maxscore information is the WAND approach [Broder et al., 2003, Tonellotto et al., 2010].

[Shan et al., 2012] demonstrates that the new block-max index can work with both maxscore and WAND algorithms to further speed up query processing.

[Fontoura et al., 2011] contains a fairly recent survey of major query processing algorithms under both DAAT and TAAT approaches.

[Strohman and Croft, 2007] also studies efficient query evaluation for memory-resident indexes.

References

[Broder et al., 2003] Broder, A. Z., Carmel, D., Herscovici, M., Soffer, A., and Zien, J. Y. (2003). Efficient query evaluation using a two-level retrieval process. In *CIKM*, pages 426–434.

[Fontoura et al., 2011] Fontoura, M., Josifovski, V., Liu, J., Venkatesan, S., Zhu, X., and Zien, J. Y. (2011). Evaluation strategies for top-k queries over memory-resident inverted indexes. *PVLDB*, 4(12):1213–1224.

[Jonassen and Bratsberg, 2011] Jonassen, S. and Bratsberg, S. E. (2011). Efficient compressed inverted index skipping for disjunctive text-queries. In *ECIR*, pages 530–542.

[Shan et al., 2012] Shan, D., Ding, S., He, J., Yan, H., and Li, X. (2012). Optimized top-k processing with global page scores on block-max indexes. In *WSDM*, pages 423–432.

[Strohman and Croft, 2007] Strohman, T. and Croft, W. B. (2007). Efficient document retrieval in main memory. In *SIGIR*, pages 175–182.

[Strohman et al., 2005] Strohman, T., Turtle, H. R., and Croft, W. B. (2005). Optimization strategies for complex queries. In *SIGIR*, pages 219–225.

[Tonellotto et al., 2010] Tonellotto, N., Macdonald, C., and Ounis, I. (2010). Efficient dynamic pruning with proximity support. *Large-scale Distributed Systems for Information Retrieval*.

[Turtle and Flood, 1995] Turtle, H. R. and Flood, J. (1995). Query evaluation: Strategies and optimizations. *Inf. Process. Manage.*, 31(6):831–850.