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 $A \land B \land C$, the disjunctive query is $A \lor B \lor C$).

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 $A \land B \land C$, and we denote the documents in $A$’s inverted list as $C_A$. The candidates generated by the DAAD algorithm is $S_1 := C_A \cup C_B \cup C_C$. Now, can we reduce it to, say, $S_2 := C_A \cup C_B$?

Notice that the only set of candidates we will miss by using $S_2$ instead of $S_1$ 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 $\text{idf}(C) \cdot \max_{d \in C_2 \setminus \{tf(d, C)\}}$. If even this score is no larger than the currently found $k$-th highest score, then we can safely use $S_2$ instead of $S_1$.

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 $\text{idf}(t) \cdot \max_{d \in L \setminus \{tf(d, t)\}}$.

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

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 $\tau'$ 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 $\tau'$.

\footnote{It is important to note that we still allow the algorithm to access the inverted list of $C$ in the scoring phrase.}

\footnote{Find out which part is precomputed.}

\footnote{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. 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 \(L_i\)s are ordered by their maxscore in decreasing order

1. Initialize min-heaps \(H\) and \(topk\); /* weights are docID and score, respectively. We push \(k\) negative values into \(topk\) initially. */
2. \(RTLists \leftarrow \{L_1, L_2, \ldots, L_m\}\);
3. \(PTLists \leftarrow \emptyset\);
4. forall \(L_i\) do
5. \(H\).push\((L_i\text{.curPosting}().docID), L_i\text{.curPosting}().docID)\); \(L_i\text{.next}()\); /* curent posting contains all the necessary info for scoring plus the list id. */
6. while \(H\).isEmpty() \(\neq\) true do
7. \((\text{score}, \text{docID}) \leftarrow \text{calcScore}(H, PTLists)\); /* score another doc */
8. \(topk\).push\((\text{docID}, \text{score})\); /* update the top-k heap and the top-k bottom score */
9. \(\text{topk}\.pop()\);
10. \(\tau' \leftarrow \text{topk}\.peep().\text{score}\); /* update RTLists and PTLists based on \(\tau'\) */
11. \(\text{update}(\tau', RTLists, PTLists, H)\);

Algorithm 2: calcScore\((H, PTLists)\)

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

1. Initialize \(info\);
2. \((\text{info}, \text{docID}) \leftarrow H\.pop()\);
3. \(i \leftarrow \text{info}\).listID;
4. \(H\).push\((L_i\text{.curPosting}().docID), L_i\text{.curPosting}().docID)\); \(L_i\text{.next}()\);
5. while \(H\).peep().docID = \(\text{docID}\) do
6. \((\text{tempInfo}, \text{docID}) \leftarrow H\.pop()\);
7. \(\text{info}.\text{add}(\text{tempInfo})\);
8. \(i \leftarrow \text{tempInfo}\).listID;
9. \(H\).push\((L_i\text{.curPosting}().docID), L_i\text{.curPosting}().docID)\); \(L_i\text{.next}()\);
10. \(\text{score} \leftarrow \text{calcScore2}(\text{docID}, \text{info}, PTLists)\);

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 \(\tau'\) value and avoid calling the \text{update} function if \(\tau'\) didn’t change. The partitioning of all \(m\) lists into RTLists and PTLists can be done incrementally.
Algorithm 3: \texttt{calcScore2}($docID$, $info$, $PTLists$)

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

\begin{algorithmic}[1]
\FORALL{$L_i \in PTLists$}
    \STATE $id \leftarrow L_i$.skipTo($docID$);
    \IF{$id = docID$}
        \STATE $info.add(L_i$.curPosting$)$;
    \ENDIF
\ENDFOR
\STATE $s \leftarrow$ compute score of $docID$ based on $info$;
\end{algorithmic}

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

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

\begin{algorithmic}[1]
\STATE $upperBound \leftarrow 0$;
\FOR{$i = m$ \textbf{to} 1}
    \STATE $upperBound \leftarrow upperBound + L_i$.maxscore$;$
    \IF{$upperBound \geq \tau'$}
        \STATE \textbf{break};
    \ENDIF
\ENDFOR
\STATE $RTLists \leftarrow \{L_1, \ldots, L_i\}$;
\STATE $PTLists \leftarrow \{L_{i+1}, \ldots, L_m\}$;
\STATE Remove items in $H$ that came from a list now in $PTLists$;
\end{algorithmic}

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.

\[\text{We denote their postings lists as $RTLists$ in the algorithms.}\]
Table 1: A Running Example ($k = 2$ and Each Posting only Contains ($docID : tf$))

<table>
<thead>
<tr>
<th>term</th>
<th>maxscore</th>
<th>postings</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>8</td>
<td>$(D_1 : 2), (D_2 : 8), (D_4 : 2)$</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
<td>$(D_1 : 1), (D_4 : 4), (D_{10} : 1), (D_{11} : 4), \ldots$</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>$(D_2 : 1), (D_3 : 2), (D_4 : 1), (D_{10} : 2), (D_{11} : 2), \ldots$</td>
</tr>
</tbody>
</table>

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 al-
gorithm 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].
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


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.

memory. In SIGIR, pages 175–182.


proximity support. Large-scale Distributed Systems for Information Retrieval.