Efficiency Optimizations for Interpolating Subqueries

Marc-Allen Cartright and James Allan
CIIR
Dept. of Computer Science
Univ. of Massachusetts
Amherst, MA 01003
{irmarc,allan}@cs.umass.edu

ABSTRACT

A large class of queries can be viewed as linear combinations of smaller subqueries. Additionally, many situations arise when part or all of one subquery has been preprocessed or has cached information, while another subquery requires full processing. This type of query is common, for example, in relevance feedback settings where the original query has been run to produce a set of expansion terms, but the expansion terms still need to be processed. We investigate mechanisms to reduce the time needed to process queries of this nature.

We use RM3, a variant of the Relevance Model scoring algorithm, as our instantiation of this arrangement. We examine the different scenarios that can arise when we have access to the internal structure of each subquery. Given this additional information, we investigate methods to utilize this information, reducing processing costs substantially. Depending on the amount of accessibility we have into the subqueries, we can reduce processing costs over 80% without affecting the score of the final results.

Categories and Subject Descriptors: H.3.3 Information Search and Retrieval: Relevance Feedback, Retrieval Models

General Terms: Algorithms, Performance

Keywords: query optimization, pseudo-relevance feedback, query modeling

1. INTRODUCTION

Recent research in information retrieval has recognized the utility of structure in queries [14, 12, 7, 8, 3, 9]. In addition to multiple researchers recognizing this characteristic, various organizations have leaded efforts towards researching the use of structure in queries, such as INEX[1] and TREC's Entity Track[2]. Assuming that queries have some structure has become the expectation in retrieval; therefore structure must be expected in query optimization. Towards this end, we look to improve query processing on a recurring pattern in information retrieval models: linear combinations.

Linear combinations of scores lie at the heart of many well-known models in information retrieval. The class of probabilistic models contains, or easily translates to, linear combinations of unigrams or n-grams. The Language Model [13] is technically a ranking based on joint probability of the query terms $q_1 q_2 \dots q_k = Q$ occurring in document D; however in order to avoid underflow the scores produced are the sum of the log-probabilities: $Score(Q, D) = \sum_{q \in Q} \log P(q|D)$. A term-order aware extension to the Language Model, the Sequential Dependence Model, uses the weighted geometric mean to combine the unigram, ordered, and unordered components of a given query [12]. When we take the logarithm of these quantities, the entire equation again reduces to a series of sums over log-probabilities. The Relevance Model [10], after generating the estimates for the likelihood of a term w coming from relevance model R, denoted $P(w|R)$, is a linear combination of log-probabilties as well: $\sum_{w \in V} P(w|R) \log P(w|D)$. Likewise, the binary independence models, most famously the BM25 model [15], are also linear combinations of term weights.

If we look to vector space models, we can observe a similar phenomenon. The standard cosine similarity scoring function takes the inner product of D and Q — which in turn is a sum of products between the two vectors [17]. The Rocchio algorithm for relevance feedback [16] creates an interpolation between the components of each query term:

$$q'_j = \alpha \cdot q_j + \beta \cdot \frac{1}{\|R\|} \sum_{D_i \in R} d_{ij} - \gamma \cdot \frac{1}{\|\bar{R}\|} \sum_{D_i \in \bar{R}} d_{ij}$$

These new weights are then used to provide a "more complete" vector representation of the information need. In implementation, the score consists of calculating the dot product, as before. There are numerous other examples of established models that operate as linear combinations of various components, and linear combination is often one of the first methods tried when adding new information into existing models.

The major contribution of this work is to provide optimization mechanisms to interpolating subquery components, depending on what we can assume about the subqueries. Given the increasing use of structure in retrieval, optimization algorithms should operate on queries as structured objects. We focus our efforts on linear combinations as they are a basic structure used throughout the field.

[1] http://www.inex.otago.ac.nz/
[2] http://ilps.science.uva.nl/trec-entity/

CIKM '11

2. RELATED WORK

Query processing optimization has been a constant activity in information retrieval since the field's inception. Many strategies have been developed for various models, providing different guarantees. While we cannot cover the entire history of optimization in IR, some prior work has more directly influenced the work here.

The use of an inference network as a retrieval model was introduced by Turtle in Croft in the early 1990s [22, 23]. Their analyses showed that a significant number of existing retrieval models at the time could be efficiently represented in the inference network framework, a form of directed graphical model, making it capable of representing a wide range of retrieval models. This framework is the inspiration for the graphic representation of queries used here, which affords us an efficient way to manipulate the query structure.

Turtle and Flood developed the MAXSCORE optimization for early-termination in both document-at-a-time (daat) and score-at-a-time (saat) query evaluation strategies. Following on this work, Strohman et al. combined the "topdocs" caching method of Brown [6] with Turtle and Flood's MAXSCORE algorithm, producing a hybrid algorithm that outperformed both prior techniques [21]. We make use of this version of the algorithm several times here.

In work focusing on saat index organization, Anh and Moffat extended the work of pruned query evaluation to impact-sorted indexes [2]. The document scores are first binned and truncated to integers, allowing for better compression and faster query evaluation. They then use an algorithm to limit the size of the accumulator table during scoring. They maintain a top candidate list in addition to the accumulator table, and use score bounds to determine when they can stop adding candidates, as well as stop tallying scores. Their paper also provides an excellent overview of index models and optimization strategies to date. Our approach here is not directly comparable to their work, as we study a different index organization, however several of our algorithms would work in conjunction with optimizations designed for saat-organized indexes.

Research in the area of query expansion includes Billerbeck and Zobel, who conduct several rounds of experiments examining the use of auxiliary data structures for improving the efficiency of query expansion [5, 4]. They discovered that using short summaries of documents significantly reduced the time needed to analyze the documents used for feedback, which they considered to be the largest bottleneck during automatic expansion. In contrast, our concern is not with generating the expansion query - the techniques shown here are applied after such generation. In theory we could use any method we liked to generate the expansion terms. The methods described by Billerbeck and Zobel should smoothly integrate with the merging techniques shown here, although experiments to prove this hypothesis are beyond the scope of this work.

Cartright et al. first look into the problem of specifically optimizing the Relevance Model [11]. They considered a document-document similarity matrix to serve as the offline store for supplemental data. They then tried to reduce the size of this matrix in order to improve its scalability to larger scale collections. Their results indicate that both static (i.e., index time) and dynamic (i.e., query time) pruning techniques have potential to dramatically reduce the size of the matrix with minor negative impact on the effectiveness of the original Relevance Model. Their approach attempted to directly optimize retrieval scores via document-document similarity rather than manipulating the structure of the query components, as we do here.

Collection	# docs (10^6's)	terms (10^6's)	unique terms (10^3's)
AQUAINT	1.0	484	892
GOV2	25.2	2,552	62,433
ClueWeb-B	50.2	39,834	468,380

Table 1: Statistics on the collections used in experiments.

3. APPROACH

In this section we describe the different algorithms we test in our experiments. We begin with data, software, and evaluation in order to provide context necessary for the remaining discussion in this section.

3.1 Data and Software

We conduct our experiments on three collections, as shown in Table 1. The AQUAINT collection refers to the dataset and 50 queries from the TREC 2005 Robust Track. GOV2 uses the dataset and 150 automatic run queries from the Adhoc task of the TREC 2006 Terabyte Track. Clue-B refers to the ClueWeb Category B sub-collection and 50 queries used in the Adhoc task of the TREC 2009 Web Track.

We conduct all retrieval experiments using the Galago search engine[3]. We generate all on-disk data structures, such as the indexes and the topdocs lists [21] described in later sections, using the TupleFlow distributed processing framework [19].

3.2 Evaluation

Our primary concern is improving query processing time. We perform two separate kinds of evaluation to measure the effect on processing time. First, we record the number of times we call SCORE on a term scoring node, which is the only construct in our model that directly iterates over data stored on the disk. Since the optimizations described here focus on reducing this factor, we get an upper bound of how effective these algorithms can be in reducing processing load. Second, we record the actual elapsed time of each query to determine the improvement in processing time in a live system. We use the first evaluation as the focus of our initial analysis of the algorithms, and we discuss the second type of evaluation in Section 4.4.2.

Some techniques presented here guarantee that if a document appears in R, the final ranked list, the score is the same as the unoptimized version. We call an algorithm with this property *score-accurate to rank k*. Prior work has referred to a similar property, *rank-safe up to k*, meaning the ranks of documents up to rank k are unchanged from the original ranking function [24]. As defined, these two properties are unrelated: an algorithm may exhibit the *score-accurate* property, but not the *rank-safe* property, and vice-versa. This occurs, for example, when an algorithm only fully scores a subset of the entire candidate list. A document

[3]http://galagosearch.org

in the original R may not be scored by the algorithm, therefore is not in the new R, however all documents appearing in the new R have been scored correctly. The REWIND algorithm, described in Section 3.4.2, behaves in this fashion. For the remainder of this work, we assume r is the number of documents requested, so we abbreviate *score-accurate to rank r* to *score-accurate*, and *rank-safe up to r* to *rank-safe*.

In the cases where an optimization is score-accurate and rank-safe, we do not report the effect on retrieval effectiveness. Otherwise, we report Mean Average Precision (MAP) as implemented by the `trec_eval` program[4]. In calculating the statistical significance between score count samples, we use a paired randomization test, as described by Smucker et al. [18]. When determining statistical significance of the real time measurements, we use a standard randomization test to maintain tractability. In both cases we set our sample size to 10^6.

3.3 Representation

We choose to focus on the RM3 scoring function [1] as our exemplar scoring model for this work, for several reasons: 1) previous research has shown it to be effective in practice [1]; 2) it interpolates only two subquery components, making it a canonical case for our study; and 3) the internals of both subqueries also have simple structure, affording easy manipulation.

RM3 is a variant of the Relevance Model that interpolates the Language Model [13] and Relevance Model [10] scores for a given query Q and document D:

$$Score_{RM3}(Q,D) = \lambda Score_{LM}(Q,D) + (1-\lambda) Score_{RM}(Q,D)$$

Clearly if $\lambda \in \{0,1\}$, then we can simply drop the zero-weighted subquery, and only evaluate the remaining subquery. However in cases where $0 < \lambda < 1$, both subqueries matter, and we must evaluate both. We focus on this latter case.

Typical implementations of RM3, such as the implementation in Indri [20], construct an interpolated query after an initial round of retrieval and submit the entire revised query for scoring. The algorithms described here are applied after the expansion terms have been selected. Instead of resubmitting the revised query, including the initial query as a subquery, we intervene to leverage structure in this expanded query in order to improve performance.

Given the point of intervention, we slightly recast the formula above to specify the components involved. The original query we label P, which is the processed subquery at this point. The expansion terms we label U, the unprocessed part of the query. Therefore we reformulate the RM3 scoring function to reflect this:

$$Score_{RM3}(Q,D) = \lambda Score_{LM}(P,D) + (1-\lambda) Score_{RM}(U,D)$$

Note that in order to create U, the system must have generated an initial ranked list based solely on P; we call that list R_P. We allow the system to access this initial ranked list during processing.

In order to organize the remaining discussion, we represent the query structure as a network of interconnected nodes, similar to an Inference Network [22]. As an example, suppose P is the query *hydrogen energy*, and we construct U using 3 expansion terms, *science*, *nuclear*, and *research*. Assume that each term comes with an associated weight. We label the associated weights as follows: the i^{th} term in subquery P has associated weight p_i, likewise the j^{th} term in subquery U has associated weight u_j. Let $\pi(t,D)$ and $\upsilon(t,D)$ represent the scoring functions of a single term given a document D and term t for subqueries P and U, respectively. Mathematically, the score of a document D for this query is:

$$\begin{aligned} Score(Q,D) &= \lambda Score_{LM}(P,D) + (1-\lambda) Score_{RM}(U,D) \\ &= \lambda(p_1\pi(\text{hydrogen},D) + p_2\pi(\text{energy},D)) \\ &+ (1-\lambda)(u_1\upsilon(\text{science},D) + u_2\upsilon(\text{nuclear},D) \\ &+ u_3\upsilon(\text{research},D)) \end{aligned} \tag{1}$$

In choosing RM3 as our scoring function, π and υ are in fact the same term scoring function: the log of the smoothed term likelihood of t in D.

The graphical representation of this query is shown in Figure 1. The construction provides us with a clean way of expressing the query visually. The nodes at the bottom of the tree are *term scoring nodes* - they represent the scoring functions π and υ in Equation 1. The nodes up the tree are all *combining nodes* - they combine the incoming scores according to the weights along the incoming edges. In our case, all combining nodes represent a weighted sum over the contributed scores. Assuming N is a combining node, $\hat{N}$ denotes all nodes directly under node N, excluding N. In implementation, the term scoring nodes act as iterators over the posting list of the given term, and the combining nodes act as meta-iterators that ensure the scoring over documents proceeds correctly.

The processing model uses document-at-a-time processing, meaning each document is scored in its entirety before another document is scored. Under this model, we construct the network once, and iterate over all documents, having each term scoring node score each document (a term scoring must produce a score for any document), combine the scores, then move to the next document. We assume all iterators are capable of the following functions: 1) *score* a given document, 2) *tell* the document id that the iterator is currently on in the posting list, 3) *reset* to the beginning of its posting list, and 4) *move* to/past a particular document id in its posting list[5]. Combining nodes simply pass the command forward to all nodes under them. For example, the statement 'We reset P ...' means that the term scoring iterators under P are all reset to the beginning of their respective posting lists.

During description and analysis of the algorithms, we will also consider the set of documents that appear in the posting list of a given term. We denote this as $\Delta(N)$, where N is a node in the network. If N is a combining node, then $\Delta(N) = \bigcup_{M \in \hat{N}} \Delta(M)$.

3.3.1 Visibility

Given the graphical representation of a query, we now assume that we have varying amounts of information about each combining node that we can leverage. We define an additional property over the combining nodes, known as its

[4] http://trec.nist.gov

[5] All posting lists are sorted in increasing document id order, therefore this function can be easily implemented as a less-than conditional.

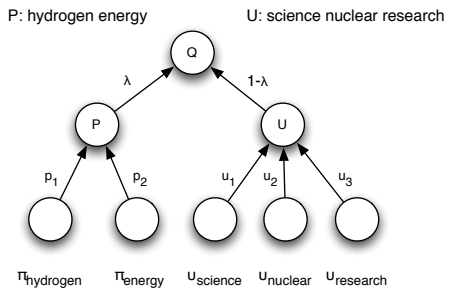

Figure 1: Graphical representation of an example query.

visibility, which has two values: *accessible* and *inaccessible*. We say a combining node N is accessible if we have the capability to examine and modify the weights and nodes in the subtree of N. We denote this as $A(N)$. If N is inaccessible, denoted $\bar{A}(N)$, we cannot examine or modify the internals of N.

We examine 4 distinct cases, shown in Figure 2. In the first case, we have $\bar{A}(P)$ and $\bar{A}(U)$, meaning we cannot directly observe or modify the structure of either subquery. Treating a node as inaccessible guarantees that the semantics of the subquery are not inadvertently changed, which is desirable in some cases. This situation may occur in federated search or distributed search, where a query comes from an external source, and the ability of the system to restructure the query is restricted. This may also occur when both P and U represent scoring nodes that the system cannot modify, for example if both subqueries are phrases. In the second case, we have $A(P)$ and $\bar{A}(U)$ and in the third case $\bar{A}(P)$ and $A(U)$. Since we assume that P has undergone some amount of processing whereas U has not, we do not assume the optimizations available between cases 2 and 3 will be the same. These cases occur whenever we encounter a subquery the system may understand (e.g., a subquery that is also a linear combination, such as a Language Model query), but the other subquery cannot be understood or is implemented externally, for example if the node represents a function that calculates a score for D based on geolocation data. We can still use the node to score documents, but we have no way of reorganizing the implementation of the node. Finally, we have $A(P)$ and $A(U)$ which is the fourth case. This represents the fortunate case where the system can manipulate both subqueries. As shown earlier, this case occurs frequently across various text-based retrieval models. We do not consider the case when $\bar{A}(Q)$, as that leaves us with no information about the query structure, therefore the methods described here do not apply.

3.3.2 Maxscore

In addition to the standard linear combining nodes in our network, we also implement the MAXSCORE algorithm described by Turtle and Flood [24] as a combining node. Fundamentally a MAXSCORE node is a combining node that performs extra bookkeeping to determine if it can stop combining scores early. If it stops early, then the candidate document would not have passed the threshold score (the lowest score in the list of top r documents so far) and it returns the incomplete score early. Otherwise the node completes

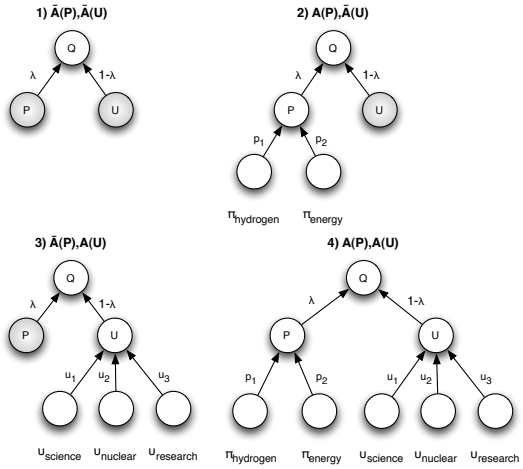

Figure 2: 4 cases under investigation, with varying amounts of accessibility. Shaded nodes are inaccessible.

scoring and returns the correct score. We denote a MAXSCORE combining node N as N^M. We define a function, MAKE-MAXSCORE, which accepts a regular combining node and produces the MAXSCORE-enabled version of that node. Mathematically, the two node types will return the exact same documents and scores for the number of documents requested (r). While we may replace any combining node in the network with its MAXSCORE equivalent, we consider the direct interaction of multiple MAXSCORE nodes outside the scope of this work.

We also augment MAXSCORE in a similar manner as Strohman et al. by including *topdocs lists* for the term scoring nodes [21]. The topdocs implementation in Galago operates similar to the implementation in Indri. Every posting list over length l has its best b scoring documents stored in a short list, known as its topdocs list. When MAXSCORE accesses these lists, it partially scores the union of all available topdocs lists, allowing the algorithm to immediately lower its threshold to filter candidates, as well as lowering the potential on each term scoring node with a topdocs list. Both l and b are set to 1000 for our experiments.

In Galago, the topdocs are kept as a separate posting list file in the index, and are only accessed when requested by the retrieval model. While this organization may not be as real-time efficient as Indri, we believe this creates a better interface for future experimentation with such data structures where supplemental information can be made available on demand. We implemented the MAXSCORE node to check for topdocs lists. Therefore if a term scoring node has a topdocs list available, the MAXSCORE node will utilize that extra information.

We now describe the experimental algorithms we test in each scenario. For the following discussion, we assume R is the final ranked list returned by the algorithm.

3.4 $\bar{A}(P)$ and $\bar{A}(U)$

We begin with the first case, where both P and U are inaccessible.

3.4.1 Fusion

FUSION is the first, and simplest, of the algorithms we present. The intuition behind this algorithm is simple: RM3 scores using both P and U. Instead, FUSION only makes use of whatever R_P provides, and ignore P. The pseudocode for FUSION is shown below. We assume R_P and R_U act as maps from the documents returned to their scores, for notational convenience. The algorithm begins by evaluating U, producing ranked list R_U (lines 1-4). To guarantee that every document receives some kind of score, we establish lower bounds for each list (lines 5 and 6). These bounds acts as the default score for any document not explicitly in the corresponding list. We then take the linear interpolation of R_P and R_U using parameter λ (lines 7-14). Finally, we return the highest-scoring r items from this merge (lines 15-19).

FUSION(R_P, U, λ, r)
1 $R_U \leftarrow$NEW-QUEUE
2 **foreach** $d \in \Delta(U)$
3 INSERT($R_U, \langle d, \text{SCORE}(U, d)\rangle$)
4 TRIM(R_U, r)
5 $min_P \leftarrow \min_{d \in R_P} R_P[d]$
6 $min_U \leftarrow \min_{d \in R_U} R_U[d]$
7 $A \leftarrow$ NEW-MAP()
8 **foreach** $d \in R_P$
9 $A[d] \leftarrow \lambda R_P[d] + (1-\lambda) min_U$
10 **foreach** $d \in R_U$
11 **if** $d \in A$
12 $A[d] \leftarrow A[d] + (1-\lambda)(R_U[d] - min_U)$
13 **else**
14 $A[d] \leftarrow \lambda min_P + (1-\lambda) R_U[d]$
15 $R \leftarrow$ NEW-QUEUE()
16 **foreach** $d \in A$
17 INSERT($R, \langle d, A[d]\rangle$)
18 TRIM(R, r)
19 **return** (R)

The FUSION algorithm is simple, requires no further disk-accesses after evaluation, and two linear passes through R_P and R_U, which are typically much smaller than the number of documents evaluated to produce the corresponding lists. Note that the only time we call the SCORE function is at the beginning, when we populate R_U (line 3). We do not score any document against P, whereas the standard RM3 algorithm would. This is where we receive the gain in efficiency over RM3. This method also has the advantage that it can be applied across virtually all scoring and processing strategies, as long as the final scores can be combined independently of each other (i.e. the final score of a document does not rely on any other score in the same list).

FUSION is neither score-accurate nor rank-safe: a document that does not appear in R_P will not receive the correct score - instead it will receive the lowest available score in R_P. This in turn can cause a change in the ordering of documents in R. Consequently all documents not in R_P are overestimated if they appear in R_U. The converse is true as well. Also, the ability of this method to reduce the number of accesses is limited – as the number of expansion terms grows, the percent of processing that FUSION will save relative to processing the whole query will decrease.

3.4.2 Rewind

In an effort to improve correctness among the final scores, we present another algorithm, which caches R_P and makes use of the scores in it when possible; otherwise we use P to score the document. We call this algorithm REWIND, since we rewind P, and use it to score documents as requested by U. The pseudocode for REWIND is shown below.

REWIND(P, R_P, U, λ, r)
1 RESET(P)
2 R $\leftarrow$ NEWQUEUE()
3 **foreach** $d \in \Delta(U)$
4 **if** $d \in R_P$
5 $score \leftarrow \lambda R_P[d] + (1-\lambda)$SCORE($U, d$)
6 INSERT($R, \langle d, score\rangle$)
7 **else**
8 MOVETO(P, d)
9 $score \leftarrow \lambda$SCORE(P, d) $+ (1-\lambda)$SCORE(U, d)
10 INSERT(R, $\langle d, score\rangle$)
11 TRIM(R, r)
12 **return** R

The REWIND algorithm is not as efficient as FUSION; in addition to calling SCORE with node U, it also will frequently call SCORE using node P (line 9). However iteration is directed by U (line 3), so REWIND still makes less SCORE calls than the original RM3 algorithm.

REWIND is a score-accurate algorithm but it is not rank-safe. Consider the set $\Delta(P) - \Delta(U)$. These are documents in the posting list(s) of P that are not in U. They will never be considered as candidates, therefore they will never receive a score, and have no chance of making it into R, the final ranked list. Therefore, all of the scores returned in R are correct, however some documents that should be in R may be omitted.

3.5 $\mathbf{A(P)}$ and $\mathbf{\bar{A}(U)}$

We now assume that P is accessible, therefore we have access to its internal structure. The first modification we make is to replace P with P^M, its MAXSCORE equivalent, to improve evaluation over P's subtree. However we can go even further, and incorporate U directly into P^M's subtree. We do not need access to U to do this; we only need to take care to properly adjust the weight of U when we add it. We call this algorithm MAX-PLUS.

3.5.1 Max-Plus

Our plan is to add P to the subtree of U^M. However we must take care to properly adjust the weights when adding P. Fortunately, we have access to the weights associated with $\hat{P}^M$ as well as λ, our interpolation weight between the two components. Using these we can derive the appropriate weight for U. Let W_U be the desired weighting for U, $W_{P^M} = \sum_{i=|P^M|} p_i$, $W = W_{P^M} + W_U$ and by definition $\lambda + (1-\lambda) = 1$. We know the ratios of $\lambda : 1$ and $W_{P^M} : W$ are equivalent:

$$\frac{\lambda}{1} = \frac{W_{P^M}}{W} \rightarrow \lambda = \frac{W_{P^M}}{W}, \quad \text{implying that}$$
$$\lambda W = W_{P^M} \rightarrow W = \frac{W_{P^M}}{\lambda}$$

Substituting into the ratio for W_U:

$$\frac{1-\lambda}{1} = \frac{W_U}{W} \rightarrow 1-\lambda = W_U \frac{\lambda}{W_{P^M}}, \quad \text{we see that}$$

$$W_U = (1-\lambda)\frac{W_{P^M}}{\lambda}$$

meaning we can represent the required weight for U in terms of the weights of $\hat{P}^M$ and λ. The conversion of the query example is shown graphically in Figure 3.

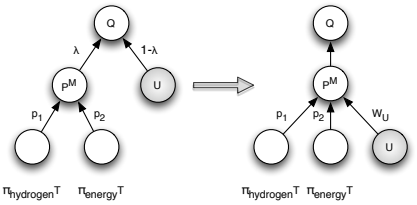

Figure 3: Converting the standard maxscore expansion to maxscore-plus.

3.6 $\bar{A}(P)$ and $A(U)$

We now assume that U is accessible but P is not. As before, we replace U with U^M, to prune as much as we can over U's subtree. We can use the MAX-PLUS modification to incorporate P directly into the subtree of U^M. The calculation for W_P proceeds in the same fashion as for W_U earlier, therefore $W_P = \lambda(1-\lambda)^{-1}W_{U^M}$. To differentiate this version of the algorithm from MAX-PLUS, we use the label MAX-PLUS-U in the results section.

3.6.1 Warmup

Unlike the case of $A(P)$ and $\bar{A}(U)$, we can now make use of R_P to modify the operation of U. We would like to avoid adding P into the subtree of U^M, to save the cost of iterating over the posting lists in P again. Instead, we only use R_P to populate the internal candidate list of U^M; in other words we "warm up" the cache in U^M. From this point forward, we use the FUSION algorithm to produce final results. We define the SETBOUNDS function, which populates the candidate list of U^M with R_P. The pseudocode for WARMUP is shown below.

WARMUP(R_P, U, λ, r)
1 $U^M \leftarrow$ MAKE-MAXSCORE(U)
2 SETBOUNDS(U^M, R_P, λ, r)
3 **return** FUSION(R_P, U^M, λ, r)

3.7 $A(P)$ and $A(U)$

We come to the situation where we can observe the internal structure of both P and U.

3.7.1 Max-Flat

We continue in the vein of MAX-PLUS and rewrite the query in order to flatten it into a single layer. Figure 4 shows the flattening of our query example from earlier. As before, we replace Q with Q^M to use pruning where possible. We remove nodes P and U entirely, and report scores directly to Q^M. Note that P and U themselves do not contribute to the scoring tally for our measurements. Therefore their removal does not affect the number of nodes being tallied; any change in the count will come from pruning.

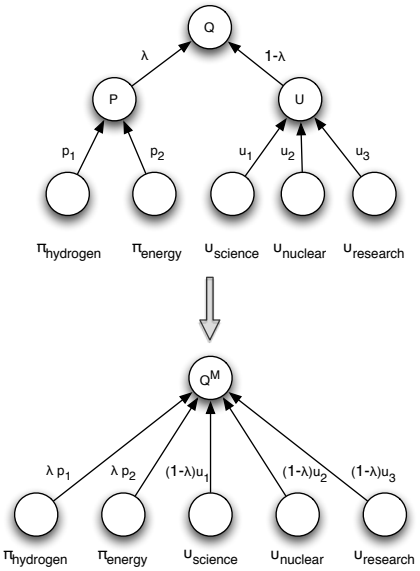

Figure 4: Flattening the query tree.

This assumption, that our components P and U may be expressed as linear combinations themselves, holds true for many of the popular retrieval models studied today. Therefore while it is a strong assumption, it also applies to a large portion of models studied to date.

4. EXPERIMENTS AND RESULTS

We vary three parameters when performing experiments:

Interpolation Weight The weight λ determines the importance of each subquery. We investigate three distinct values of λ: 0.2, 0.5, and 0.8 to examine how changing the importance of the subqueries affects each of the algorithms.

Number of Expansion Terms The ratio of terms between P and U will most likely impact the effectiveness of the algorithms. Let $\tau = |\hat{U}|$, the number of terms in the expansion. We use values of 10 and 100 for τ to examine minor and major imbalance.

Number of Items Requested We vary r, the number of documents requested in R, to determine how the algorithms are affected by needing to maintain larger candidate lists. This should affect the non score-accurate algorithms the most. We use values of 100 and 1000 to observe this effect.

We present the main efficiency results in Table 2. Each row provides the results for a particular parameter setting in a given collection, and a column is the performance of a method over the different parameter settings. For each parameter setting, we provide the average number of times the SCORE function was called, and the percent change from the average for the RM3 method. In all cases, the *default* setting indicates the following parameter values: $r = 100$, $\lambda = 0.5$, and $\tau = 10$. We include an RM3 run that makes

AQUAINT	RM3	RM3+	Rewind	Fusion	Warmup	Max-Plus-U	Max-Plus	Max-Flat
Default	8515351	-4.4†	-5.1	-23.6	-50.6	-68.3	-7.1†	-85.3
$r = 1000$	8515351	-4.4†	-5.2	-23.6	-26.6	-64.6	-7.1†	-79.1
$\tau = 100$	104815675	-1.6†	-0.04	-2.5	-5.4	-46.8	-1.5†	-61.9
$\lambda = 0.2$	8515351	-3.0†	-5.1	-23.6	-66.5	-73.1	-2.7†	-80.0
$\lambda = 0.8$	8515351	-86.5	-5.1	-23.6	-32.0	-69.4	-92.3	-87.6
GOV2	RM3	RM3+	Rewind	Fusion	Warmup	Max-Plus-U	Max-Plus	Max-Flat
Default	193233121	-7.2	-7.0	-29.1	-63.3	-56.5	-9.0	-88.2
$r = 1000$	193233121	-0.6†	-7.0	-29.1	-40.2	-53.4	-9.0	-82.1
$\tau = 100$	2351713435	-2.8	-0.022	-3.3	-15.7	-50.8	-2.9	-75.2
$\lambda = 0.2$	193233121	-2.4	-7.0	-29.1	-76.4	-74.7	-1.5	-78.0
$\lambda = 0.8$	193233121	-86.7	-7.0	-29.1	-41.1	-54.5	-94.2	-87.6
CLUE-B	RM3	RM3+	Rewind	Fusion	Warmup	Max-Plus-U	Max-Plus	Max-Flat
Default	400366062	-23.7	-10.4	-26.5	-63.2	-58.0	-27.1	-90.7
$r = 1000$	400366062	-23.7	-10.5	-26.5	-42.1	-57.0	-27.1	-90.7
$\tau = 100$	4428706966	-11.6	-2.5	-2.5	-14.7	-58.3	-14.8	-81.2
$\lambda = 0.2$	400366062	-13.3	-10.5	-26.5	-73.9	-81.2	-8.5	-87.6
$\lambda = 0.8$	400366062	-68.9	-10.5	-26.5	-44.7	-58.1	-90.6	-93.6

Table 2: Results over AQUAINT, GOV2, and ClueWeb-B, using 36, 150, and 50 queries, respectively. The † indicates a change that is *not* statistically significant. The number in the RM3 column is the number of score requests under the unmodified algorithm. The numbers in the remaining columns are the percent change relative to the the unmodified RM3 model. We calculate this as $-(A - B)/A$, where A is RM3 and B is the algorithm in question.

use of MAXSCORE (RM3+), for comparison. In the graphical representation, we accomplish this by replacing node Q with Q^M, therefore we use the MAXSCORE algorithm directly on nodes P and U.

The results from the three datasets look largely equivalent, suggesting that the methods described here will retain their ability to improve over the baseline as the collection size increases. Both RM3+ and MaxPlus seem to generally perform better on the Clueweb-B dataset; analysis of this phenomenon reveals that the proportion of queries actively optimized (i.e. pruned) is much higher in Clueweb-B ($> 45\%$ in both cases) than in the other two collections ($< 23\%$). This in turn produces a larger impact on the average SCORE count.

Method	Default	$\tau = 100$	$r = 1000$	$\lambda = 0.2$	$\lambda = 0.8$
Baseline	0.0827	0.0827	0.1565	0.0827	0.0827
RM3	0.1153	0.1177	0.1984	0.1000	0.0991
Fusion	0.0978	0.1096	0.1888	0.0911	0.0912
Rewind	0.1153	0.1177	0.1945	0.1000	0.0993
Warmup	0.0824	0.0958	0.1678	0.0777	0.0793

Table 3: MAP results over the AQUAINT collection. Only methods that are non-safe are shown. The baseline method is the query-likelihood model, shown for comparison.

4.1 $\bar{A}(P)$ and $\bar{A}(U)$

The behavior between the REWIND and FUSION algorithms is consistent between the two collections; FUSION provides a significant decrease in the number of scores requested, while REWIND only reduces the evaluation cost by less than 10%. However we must also consider MAP scores for these algorithms. Table 3 shows the MAP scores produced by the non-safe[6] algorithms across the different parameter settings. Generally, the FUSION algorithm only provides a relatively small increase in performance over the baseline method (Language Model), and only in the case of $r = 1000$ does the performance of FUSION approach that of RM3.

Conversely, REWIND seems to stay true to the original scoring model, except in two cases. When $r = 1000$, the score is slightly lower than RM3. Analysis of the mean precision at different ranks shows identical precision scores until past rank 200. When $\lambda = 0.8$, the score is slightly higher than RM3, most likely due to the increased weight on U. Hence, more relevant documents would come from U, which are all correctly scored using REWIND.

When $\tau = 100$, we also see an interesting result. The improvement from REWIND is minimal ($> 1\%$), however the result is statistically significant. RM3+ produces a larger improvement, but it is not significant. Further analysis revealed that RM3+ only improved over RM3 in a few cases, but these cases drastically pulled down the average score count. In contrast, REWIND consistently outperformed RM3 on every query, but only by a small amount. In this case, the significance test indicates one outcome, while the effect size indicates the opposite outcome.

4.1.1 *Fusing Larger Lists*

Given the results above, we wonder if increasing the number of results from just U would improve results. We have no control over R_P, however we can instruct U to return a larger list of documents that we could use in merging. We run a few experiments to observe this behavior (Table 4). We do not show the average number of scores requested, as they are the same across all runs. We express the increase as a percentage of the original list, i.e. a value of 10 indicates

[6]Non-safe indicates an algorithm lacks either the rank-safe or the score-accurate property.

that we request $1.1r$ (10% more) documents to be scored, where r is the original list size.

% More	0	10	25	50	100
MAP	0.0978	0.0977	0.0984	0.0986	0.0983

Table 4: Changes in MAP when varying the number of results requested from U.

The results suggest that increasing the size of the list returned from U can provide some minor benefits, but only up to a point. The decrease when we double the size of the list indicates that around that point we stop adding useful documents to the merging procedure, and that the overestimated score from P for those documents introduces noise in R.

4.1.2 Maxscore Under the Hood

While we have no control over the internals of either subquery in this configuration, we examine how FUSION and REWIND may interact with MAXSCORE. To test this effect, we replace P and U with their MAXSCORE version where appropriate. In FUSION, we apply it to U, and in REWIND, we apply it over the cached P and U. The results over the default parameter set are shown in Table 5. We can immediately see a large improvement in efficiency over the baseline methods. This indicates that FUSION and REWIND can combine with MAXSCORE to improve other using either technique alone. We leave further analysis as future work.

Method	# Scores	% Chg.	Orig
RM3	8143117.5	0.0	-4.4
Fusion	1791606.1	-78.0	-72.4
Rewind	2352511.6	-71.1	-77.8

Table 5: Results over the AQUAINT collection, now using the MAXSCORE optimization. The rightmost column indicates the change over the same method without the optimization.

4.2 $\mathbf{A(P)}$ and $\mathbf{\bar{A}(U)}$

The MAX-PLUS algorithm is the only algorithm tested for this scenario. Under the default and $r = 1000$ parameter settings, MAX-PLUS provides a consistent minor improvement over the original algorithm. The differences are not statistically significant in the AQUAINT collection, however they are for GOV2. If the number of feedback terms is increased, the effectiveness of the algorithm decreases, which we expect — we have no control over U, therefore if its organization changes, it will have an effect on MAX-PLUS.

The most interesting result comes when $\lambda = 0.8$. Both RM3+ and MAX-PLUS perform exceptionally well. Further inspection reveals that in both cases, the bias towards P allowed for massive pruning during query processing. The only documents that have a chance to enter the candidate list are documents that contain components from P; specifically, the entire subquery. Therefore the only documents that are even considered are in $\Delta(P)$. The main difference between these algorithms is that MAX-PLUS directly integrates U into its early-termination algorithm, whereas RM3-Maxscore can only interact with the nodes opaquely. We believe this allows the MAX-PLUS algorithm to converge to a stable threshold more rapidly, resulting in the increase in performance over RM3-Maxscore.

4.3 $\mathbf{\bar{A}(P)}$ and $\mathbf{A(U)}$

We now look to the performance of WARMUP and MAX-PLUS-U. Both algorithms provide substantial improvements over the original RM3 algorithm. As anticipated, WARMUP consistently outperforms the FUSION algorithm in efficiency, where we do not pre-set the candidate list for the U node. However if we consider effectiveness (Table 3), we see that in fact WARMUP performs worse than FUSION in all cases, and in some cases performs worse than the baseline method.

In contrast, the MAX-PLUS-U method appears to consistently provide substantial gains in efficiency, while maintaining score correctness. The only noticeable drop in performance for the MAX-PLUS-U algorithm comes from increasing the number of feedback terms ($\tau = 100$), which is consistent with the behavior of the other algorithms. However unlike several of the other algorithms, MAX-PLUS-U appears to be more robust to the parameter change.

4.4 $\mathbf{A(P)}$ and $\mathbf{A(U)}$

Clearly, having access to both sub-nodes allows the most effective optimization of the entire set. The MAX-FLAT procedure consistently shaves off over 80% of the scoring requests, except when we increase the number of feedback terms by a factor of 10. Even under such conditions, MAX-FLAT still cuts over half of the scoring requests. Deeper analysis of this model shows that like MAX-PLUS and MAX-PLUS-U, having all of the term nodes managed by one MAXSCORE node allows for the fastest stabilization of the threshold scores. The only case where this is not true is when $\lambda = 0.8$, where P has so much weight that it largely overwhelms the contributions of U, allowing MAX-PLUS and RM3+ to perform on par with MAX-FLAT.

4.4.1 Modifying the Scoring Order

In the original MAXSCORE algorithm, the term scoring nodes are ordered according to the estimated lengths of their posting lists, the intuition being that if pruning occurs by the nodes processed first during a scoring pass, then moving the shortest posting lists to the front should minimize the number of documents considered. However our model has scoring nodes scaled by weights. Not only must we consider weights now, but as we rescale the weights of the nodes managed by the algorithm, as we do in the MAX-* algorithms, we often have the situation that some subset of the nodes have weights orders of magnitude larger than the others.

Considering these new factors, we explore the possibility of ordering by weight to increase pruning further. We reason as follows: If the pruning decision is contingent on the heavily weighted nodes, then if we order by length of the posting lists, this weight imbalance is ignored. Hence we tend to prune wherever we hit these heavily weighted nodes, no matter where they fall in the scoring order. Ordering by weight pushes these nodes to the front of the scoring list, removing the wasted effort of scoring nodes that do not trigger pruning. We test this hypothesis on the MAX-FLAT algorithm, and call it MAX-FLAT-W. Results are shown in Table 6 for different values of λ, and for $\tau = 100$.

Surprisingly, ordering by weight is only effective when $\lambda = 0.8$ or when $\tau = 100$. This indicates that the weight imbalance between the subquery components must be quite

Method	Max-Flat	Max-Flat-W	% Chg
$\lambda = 0.2$	1876666.2	3430090.9	+82.3
$\lambda = 0.5$	1248424.9	1795394.2	+43.8
$\lambda = 0.8$	1053798.0	615244.3	-41.6
$\tau = 100$	39932114.8	37111540.1	-7.1

Table 6: Comparing list length and weight ordering for the MAX-FLAT algorithm.

high for it to be effective. Worse performance of MAX-FLAT-W when $\lambda = 0.2$ supports this hypothesis. This weight brings the individual component weights closer to uniformity, reducing the importance of ordering the scoring list by weight.

4.4.2 *Effect on Wall-Clock Time*

We now examine changes in performance based on system time measurements. We perform 5 runs of the GOV2 collection, with the queries permuted randomly in each run to lessen any dependence on query order. We record the time to execute each query for each method, making a total of 750 samples for each algorithm. The system used is a non-dedicated cluster of 32 nodes, each node having 2 Xeon 3.2 GHz cores with 4Gb of RAM and having access to a shared disk-server, where the index resides. Therefore a single run executes on a single core in this cluster. To compute statistical significance between algorithms, we use a randomization test of 1 million samples with the sample mean as the sufficient statistic. We also determine what percentage of the queries shows some "positive effect" relative to unmodified RM3. Let $\mathcal{A}$ be the algorithm in question. If the value for $\mathcal{A}$ is less than 90% of the value for RM3 for a particular measure (either SCORE count or real time), we consider that to be a positive effect. For example, a value of 10 in the table indicates that 10% of the queries (75 of 750) had that measure reduced by at least 10% compared to RM3. Table 7 shows the mean runtimes of the samples, the percent change from RM3, and percentage of queries affected when measuring via SCORE count and real time.

Based on the results from Table 7, the algorithms seem to fall into 3 distinct groups. The strongest of these is MAX-FLAT, and to a slightly lesser degree, MAX-PLUS-U, which seem to consistently reduce both the SCORE count and the system time by a significant amount. Their average impact is high, and they cover a high percentage of the queries. The next group includes the WARMUP and FUSION algorithms. They also have a consistent impact, but their average impact is less pronounced. The final group, consisting of RM3+, REWIND, and MAX-PLUS, have average runtimes are greater than the unmodified RM3, but they still register at least a 10% measurement improvement for some number of the queries tested. Further analysis of this last group shows that when the algorithms work, they have a significant impact on both measurements, often resulting in reductions over 40%. However, as the table shows, these algorithms do not "trigger" very often, and the added logic used to continually check for a pruning opportunity results in a notable increase in execution time.

5. DISCUSSION AND CONCLUSION

We have shown that it is possible to greatly improve the efficiency of queries represented by a linear combination of

Method	Mean Time	Pct Chg	Pct. Affected	
RM3	134.8	0	Score	Time
Max-Flat	19.4	-85.6✦	100.0	100.0
Max-Plus-U	68.6	-49.1✦	87.3	86.1
Fusion	129.3	-4.1✦	98.7	41.9
Warmup	119.4	-11.4✦	100.0	60.4
RM3+	245.9	+82.4	12.0	7.1
Rewind	139.9	+3.7	20.0	44.9
Max-Plus	230.7	+71.1	22.0	13.7

Table 7: Statistics over 750 queries run over GOV2. Mean times are in seconds. The ✦ indicates statistical significance at $p \leq 0.02$. The Score and Time columns report the percentage of queries that experienced at least a 10% drop in the given measurement.

subqueries, a common mechanism for integrating partial scores in information retrieval. Our results show that when the two subqueries are inaccessible, our options for safe optimization are limited, but there are several courses of action we can take that typically have little negative effect on the accuracy of results. When we have access to the internal structures, we can manipulate the query structure further, providing greater efficiency gains while making stronger correctness guarantees. Under the correct conditions, we can reduce the scoring cost by over 50%, and in many cases by over 80%, while not affecting the ranking results at all.

Several phenomena here require further study to fully explain. All of the algorithms presented here appear to be sensitive to τ, therefore, determining the underlying cause of this sensitivity will give us even further insight into the interaction between the subquery components. Given the emergent behaviors shown by the algorithms, understanding what properties of the algorithms cause such behavior would help to predict the interactions between optimizations, queries, and collections.

Although several algorithms described provide substantial gains in efficiency, our experiments also raise new interesting questions to pursue. A simple observation from the results suggests that for different values of λ, different algorithms afford the greatest reduction in cost. Query analysis may enable us to choose which algorithm to use based on the value of λ, thereby leveraging the best possible optimization available. Also, the real-time analysis here shows that if we can cheaply predict whether a certain optimization will activate on a given query, we may be able to avoid unnecessary overhead by not trying to optimize the query in the first place.

The above results show how we can reduce the processing cost of the RM3 algorithm. However demonstrating the effectiveness of these algorithms when applied to RM3 implies their utility with other retrieval models as well. RM3 serves as an archetype of interpolated subqueries: many other retrieval models share structure similar RM3, therefore the algorithms developed here will easily transfer to those instances with little effort. Furthermore, we can look to apply these algorithms to even more diverse query types. Consider, for example, an n-gram index, where these mechanisms could significantly impact the processing time of scanning the term-level posting lists by making use of the stored n-grams. Recent research has made use of n-gram posting

lists [8, 7], therefore we can consider the pre-calculated n-gram posting list as the subquery containing information, and the rest of the query (the unigrams) as not. We may encounter subqueries that must be evaluated against different indexes, such as in desktop search as described by Kim and Croft [9] or graph-based data, such as citation analysis information. One subquery may apply to the graph data, whereas the other applies to textual information. Other applications include temporal information, precalculated statistics for document or collection-level data, and geolocation information.

In the context of web search, the optimizations discussed here would most likely not impact the head queries that are heavily cached and optimized for performance. However tail queries could benefit greatly, as the use of psuedo-relevance feedback can still improve retrieval performance for such queries. The results here indicate that we can apply pseudo-relevance feedback, among other expansion techniques, to improve ranked results without suffering from high overhead.

In the past it was sufficient to investigate optimizations for queries that were simply bags of words - therefore one could assume that each component in the query represented a single iterator over a posting list. However with the advent of structured queries that are now being applied over increasingly larger and larger sets of data, the need to respect this additional structure while designing optimizations for query processing has become evident. We believe this research moves towards addressing this need, and we intend to continue this line of research to provide even greater improvements for structured query processing.

6. ACKNOWLEDGEMENTS

This work was supported in part by the Center for Intelligent Information Retrieval and in part by NSF grant #IIS-0910884. Any opinions, findings and conclusions or recommendations expressed in this material are the author(s) and do not necessarily reflect those of the sponsor.

7. REFERENCES

[1] N. Abdul-jaleel, J. Allan, W. B. Croft, O. Diaz, L. Larkey, X. Li, D. Metzler, M. D. Smucker, T. Strohman, H. Turtle, and C. Wade. Umass at trec 2004: Notebook. In *TREC 2004*, pages 657–670, 2004.

[2] V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '06, pages 372–379, New York, NY, USA, 2006. ACM.

[3] M. Bendersky and W. B. Croft. Discovering key concepts in verbose queries. In *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '08, pages 491–498, New York, NY, USA, 2008. ACM.

[4] B. Billerbeck and J. Zobel. Techniques for efficient query expansion. 2004.

[5] B. Billerbeck and J. Zobel. Efficient query expansion with auxiliary data structures. *Inf. Syst.*, 31:573–584, November 2006.

[6] E. W. Brown. Fast evaluation of structured queries for information retrieval. In *Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '95, pages 30–38, New York, NY, USA, 1995. ACM.

[7] Hao Yan, Shuming Shi, Fan Zhang, Torsten Suel, and Ji-Rong Wen. Efficient term proximity search with term-pair indexes. In *Proceedings of the Nineteenth International Conference on Information and Knowledge Management*, pages 39–45, Toronto, Ontario, Canada, October 2010. ACM, ACM.

[8] S. Huston, A. Moffat, and W. B. Croft. Efficient indexing of repeated n-grams. In *Fourth ACM International Conference on Web Search and Data Mining*, 2011.

[9] J. Kim and W. B. Croft. Retrieval experiments using pseudo-desktop collections. In *Proceedings of the 18th ACM conference on Information and knowledge management*, CIKM '09, pages 1297–1306, New York, NY, USA, 2009. ACM.

[10] V. Lavrenko and W. B. Croft. Relevance based language models. In *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '01, pages 120–127, New York, NY, USA, 2001. ACM.

[11] Marc-Allen Cartright, James Allan, Victor Lavrenko, and Andrew McGregor. Fast query expansion using approximations of relevance models. In *Proceedings of the Nineteenth International Conference on Information and Knowledge Management*, pages 1573–1576, Toronto, Ontario, Canada, October 2010. ACM, ACM.

[12] D. Metzler and W. B. Croft. A markov random field model for term dependencies. In *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '05, pages 472–479, New York, NY, USA, 2005. ACM.

[13] J. M. Ponte and W. B. Croft. A language modeling approach to information retrieval. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '98, pages 275–281, New York, NY, USA, 1998. ACM.

[14] S. Robertson, H. Zaragoza, and M. Taylor. Simple bm25 extension to multiple weighted fields. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, CIKM '04, pages 42–49, New York, NY, USA, 2004. ACM.

[15] S. E. Robertson and S. Walker. Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In *Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '94, pages 232–241, New York, NY, USA, 1994. Springer-Verlag New York, Inc.

[16] J. Rocchio. Relevance feedback in information retrieval. In G. Salton, editor, *The SMART retrieval system: Experiments in automatic document processing*, pages 313–323. Prentice-Hall, Englewood Cliffs, NJ, 1971.

[17] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18:613–620, November 1975.

[18] M. D. Smucker, J. Allan, and B. Carterette. A comparison of statistical significance tests for information retrieval evaluation. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, CIKM '07, pages 623–632, New York, NY, USA, 2007. ACM.

[19] T. Strohman. *Efficient Processing of Complex Features for Information Retrieval*. Ph.D. dissertation, University of Massachusetts Amherst, December 2007.

[20] T. Strohman, D. Metzler, H. Turtle, and W. B. Croft. Indri: a language-model based search engine for complex queries. Technical report, in Proceedings of the International Conference on Intelligent Analysis, 2005.

[21] T. Strohman, H. Turtle, and W. B. Croft. Optimization strategies for complex queries. In *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '05, pages 219–225, New York, NY, USA, 2005. ACM.

[22] H. Turtle and W. B. Croft. Inference networks for document retrieval. In *Proceedings of the 13th annual*

international ACM SIGIR conference on Research and development in information retrieval, SIGIR '90, pages 1–24, New York, NY, USA, 1990. ACM.

[23] H. Turtle and W. B. Croft. Evaluation of an inference network-based retrieval model. *ACM Trans. Inf. Syst.*, 9:187–222, July 1991.

[24] H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *Inf. Process. Manage.*, 31:831–850, November 1995.