Efficient and Self-tuning Incremental Query Expansion for Top-k Query Processing *

Martin Theobald
Max Planck Institute for Informatics
mtb@mpi-sb.mpg.de

Ralf Schenkel
Max Planck Institute for Informatics
schenkel@mpi-sb.mpg.de

Gerhard Weikum
Max Planck Institute for Informatics
weikum@mpi-sb.mpg.de

ABSTRACT

We present a novel approach for efficient and self-tuning query expansion that is embedded into a top-k query processor with candidate pruning. Traditional query expansion methods select expansion terms whose thematic similarity to the original query terms is above some specified threshold, thus generating a disjunctive query with much higher dimensionality. This poses three major problems: 1) the need for hand-tuning the expansion threshold, 2) the potential topic dilution with overly aggressive expansion, and 3) the drastically increased execution cost of a high-dimensional query. The method developed in this paper addresses all three problems by *dynamically and incrementally merging* the inverted lists for the potential expansion terms with the lists for the original query terms. A priority queue is used for maintaining result candidates, the pruning of candidates is based on Fagin's family of top-k algorithms, and optionally probabilistic estimators of candidate scores can be used for additional pruning. Experiments on the TREC collections for the 2004 Robust and Terabyte tracks demonstrate the increased efficiency, effectiveness, and scalability of our approach.

Categories and Subject Descriptors: H.3.3: Search process.

General Terms: Algorithms, Performance, Experimentation.

Keywords: Top-k Ranking, Query Expansion, Incremental Merge, Probabilistic Candidate Pruning.

1. INTRODUCTION

Query expansion is an indispensable technique for evaluating difficult queries where good recall is a problem. Examples of such queries are the ones in the TREC Robust track, e.g., queries for "transportation tunnel disasters" or "ship losses" on the Aquaint news corpus. Typical approaches use one or a combination of the following sources to generate additional query terms: thesauri such as WordNet with concept relationships and some form of similarity measures, explicit user feedback or pseudo relevance feedback, query associations derived from query logs, document summaries such as Google top-10 snippets, or other sources of term correlations. In all cases, the additional expansion terms are chosen based on similarity, correlation, or relative entropy measures and a corresponding threshold. For difficult retrieval tasks like the above, query expansion can improve precision@top-k, recall, as well as uninterpolated mean average precision (MAP) by a significant margin (see, e.g., [20, 21]). However, in contrast to a mere benchmark setting such as TREC, applying these techniques in a real application with unpredictable ad-hoc queries (e.g., in digital libraries, intranet search, or web communities) faces three major problems [4, 5]: 1) The threshold for selecting expansion terms needs to be carefully hand-tuned, and this is highly dependent on the application's corpus and query workload. 2) An inappropriate choice of the sensitive expansion threshold may result in either not achieving the desired improvement in recall (if the threshold is set too conservatively) or in high danger of topic dilution (if the query is expanded too aggressively). 3) The expansion may often result in queries with 50 or 100 terms which in turn leads to very high computational costs in evaluating the expanded queries over inverted index lists.

This paper addresses the above three issues and provides a practically viable, novel solution. Our approach is based on the paradigm of threshold algorithms, proposed by Fagin and others [13, 14, 16, 17, 27], with a monotonic score aggregation function (e.g., weighted summation) over inverted index lists. In this framework, each per-term list is precomputed and stored in descending order of per-term scores (e.g., some form of tf·idf-style index-term weights or the term's contribution to a probabilistic relevance measure such as Okapi BM25 [31, 32]). The specific algorithm that we employ processes index lists using efficient sequential disk I/O, prunes top-k result candidates as early as possible, and terminates without having to scan the entire index lists (often after processing only fairly short prefixes of the lists). In addition, we use score distribution statistics to estimate aggregated scores of candidates and may prune them already when the probability of being in the top-k result drops below an acceptable error threshold (e.g., 10 percent).

*Partially supported by the EU within the 6th Framework Program under contract 001907 "Dynamically Evolving, Large Scale Information Systems" (DELIS)

SIGIR'05, August 15–19, 2005, Salvador, Brazil.

Our key techniques for making query expansion efficient, scalable, and self-tuning are to avoid aggregating scores for multiple expansion terms of the same original query term and to avoid scanning the index lists for all expansion terms. For example, when the term "disaster" in the query "transportation tunnel disaster" is expanded into "fire", "earthquake", "flood", etc., we do not count occurrences of several of these terms as additional evidence of relevance. Rather we use a score aggregation function that counts only the maximum score of a document for all expansion terms of the same original query term, optionally weighted by the similarity (or correlation) of the expansion term to the original term. Furthermore and most importantly for efficiency, we open scans on the index lists for expansion terms as late as possible, namely, only when the best possible candidate document from a list can achieve a score contribution that is higher than the score contributions from the original term's list at the current scan position or any list of expansion terms with ongoing scans at their current positions. The algorithm conceptually merges the index lists of the expansion terms with the list of the original query term in an incremental on-demand manner during the runtime of the query. For further speed-up, probabilistic score estimation can be used, considering score distributions and term selectivities.

The novel contributions of the paper are threefold: 1) the Incremental Merge algorithm for highly efficient query expansion, 2) its integration with top-k query processing and probabilistic prediction of aggregated scores and selectivities, and 3) a comprehensive experimental evaluation using the Robust track and Terabyte track benchmarks of TREC 2004.

2. RELATED WORK

2.1 Query Expansion

There is a rich body of literature on query expansion (see, e.g., [3, 8, 11, 18, 20, 21, 24, 30, 33, 34, 36, 37]). All methods aim to generate additional query terms that are "semantically" or statistically related to the original query terms, often producing queries with more than 50 or 100 terms and appropriately chosen weights. Given the additional uncertainty induced by the expansion terms, such queries are usually considered as disjunctive queries and incur very high execution costs for a DBMS-style query processing [4, 5]. The various methods differ in their sources that they exploit for inferring correlated terms: explicit relationships in thesauri, explicit relevance feedback, pseudo relevance feedback, query associations derived from query logs and click streams, summary snippets of web search engine results, extended topic descriptions (available in benchmarks), or combinations of various techniques. In all cases some similarity, correlation, or entropy measure between the original query terms and the possible expansion terms should be quantified (usually in some statistical manner), and a carefully tuned threshold needs to be determined to eliminate expansion candidates that are only weakly related to the original query. While such manual tuning is standard in benchmarks like TREC, it is almost black art to find robust parameter settings for real applications with highly dynamic corpora and continuously evolving query profiles [4] (e.g., intranets, web forums, etc.). This calls for *automatic* and *self-adaptive* query tuning.

[20] generate Google queries from the original query and use the summary snippets on the top-10 result page to generate alternative query formulations and performed very successful in recent TREC benchmarks. The final query expansion is a weighted combination of the original and the alternative queries. [21] uses a suite of techniques for extracting phrases and word sense disambiguation (WSD), with WordNet [15] as a background thesaurus and source of expansion candidates. Both of these techniques seem to require substantial hand-tuning for achieving their very good precision-recall performance. In the current paper, where the emphasis is on efficiency rather than effectiveness, we use a rather simple form of WSD and WordNet-based expansions. The techniques of the current paper could be easily carried over to more sophisticated expansion methods as mentioned above.

2.2 Efficient Top-k Query Processing

There is also a good amount of literature on efficient top-k query processing for ranked retrieval of text, web, and semistructured documents as well as multimedia similarity search, preference queries on product catalogs, and other applications (see, e.g., [13] for an overview). Most algorithms scan inverted index lists and aggregate precomputed per-term or per-dimension scores into in-memory "accumulators", one for each candidate document. The optimizations in the IR literature aim to limit the number of accumulators and the scan depth on the index lists in order to terminate the algorithm as early as possible [1, 2, 7, 6, 22, 25, 29, 28]. This involves a variety of heuristics for pruning potential result candidates and stopping some or all of the index list traversals as early as possible, ideally after having seen only short prefixes of the potentially very long lists. To this end, it is often beneficial that the entries in the inverted lists are kept in descending order of score values rather than being sorted by document identifiers [13, 28]. In the current paper, we assume exactly this kind of index structure.

A general framework for this kind of methods has been developed by Fagin et al. in [14] and others [16, 17, 27], in the context of similarity search on multimedia and structured data. This family of so-called threshold algorithms (TA) operates on score-sorted index lists and assumes that the score aggregation function is monotonic (e.g., a weighted summation). It uses a threshold criterion that is provably optimal on every possible instance of the data [14]. Many variations and extensions of the TA method have been developed in the last few years [9, 10, 12, 19, 23, 26, 35, 38]. The current paper builds on the particular variant developed in [35], with sequential index accesses only and a probabilistic score prediction technique for early candidate pruning, when approximate top-k results (with probabilistic error bounds) are acceptable by the application. The rationale for avoiding, minimizing, or limiting random accesses is that for very large corpora index lists are disk-resident and sequential disk I/O is an order of magnitude more efficient than random I/O (because of lower positioning latencies).

3. THE PROB-K ALGORITHM

The basic query processing on which this paper builds is the *Prob-k* algorithm developed in [35]. A query scans precomputed index lists, one for each query term, that are sorted in descending order of scores such as tf·idf or BM25 weights of the corresponding term in the documents that contain the term. Unlike the original TA method of [14], Prob-k avoids random accesses to index list entries and per-

forms only sequential accesses. The algorithm maintains a pool of candidate results and their corresponding score accumulators in memory, where each candidate is a data item d that has been seen in at least one list and may qualify for the final top-k result based on the following information (we denote the score of data item d in the i-th index list by $s_i(d)$, and we assume for simplicity that the score aggregation is summation):

- the set $E(d)$ of evaluated lists where d has been seen,
- the global $worstscore(d) := \sum_{i \in E(d)} s_i(d)$ of d based on the known scores $s_i(d)$, and
- the global $bestscore(d) := worstscore(d) + \sum_{i \notin E(d)} high_i$ that d could possibly still achieve based on the upper bounds $high_i$ for the score values in the yet unvisited parts of the index lists.

The algorithm terminates and is guaranteed to obtain the exact top-k result, when the *worstscore* of the k^{th} rank in the current top-k result is at least as high as the highest *bestscore* among all other candidates. The Prob-k algorithm implements the candidate pool using a priority queue. In addition, it can optionally replace the conservative threshold test by a probabilistic test for an approximative top-k algorithm with probabilistic precision and recall guarantees. The most flexible implementation uses histograms to capture the score distributions in the individual index lists and computes convolutions of histograms[1] in order to predict the probability that an item d has a global score above some value δ: $P[\sum_{i \in E(d)} s_i(d) + \sum_{j \notin E(d)} S_j(d) > \delta \mid S_j(d) \leq high_j]$, where $S_j(d)$ denotes the random variable for the score of d in list j. Here, δ is set to the score threshold that the candidate has to exceed, namely to the *worstscore* of the k^{th} rank in the current top-k result (coined *min-k*). When this probability for some candidate document d drops below a threshold ϵ (e.g., set to 0.1), d is discarded from the candidate set. The special case $\epsilon = 0.0$ corresponds to the conservative threshold algorithm.

Figure 1 shows pseudo code for this algorithm. The candidates and their accumulators are kept in a hash table, whereas only a small number of top candidates has to be organized in a priority queue in order to maintain the threshold condition for algorithm termination, using their *bestscore* values as priorities. The probabilistic pruning test is performed only every b index scan steps, where b is chosen in the order of a few hundred. At these points the priority queue is rebuilt, i.e., all candidates' *bestscore* values are updated taking the current $high_i$ values into account.

Algorithm 1 Prob-k(query $q = (t_1, \ldots, t_m)$):

```
1: top-k := ∅; candidates := ∅; min-k := 0;
2: //round-robin schedule for sorted accesses
3: for all index lists L_i with i=1..m do
4:    s_i(d) := L_i.getNext(); //next sorted access to L_i
5:    E(d) := E(d) ∪ {i};
6:    high_i := s_i(d);
7:    worstscore(d) := Σ_{i∈E(d)} s_i;
8:    bestscore(d) := worstscore(d) + Σ_{i∉E(d)} high_i;
9:    if (worstscore(d) > min-k) then
10:      replace min{worstscore(d')|d' ∈ top-k} by d;
11:      remove d from candidates;
12:   else if (bestscore(d) > min-k) then
13:      candidates := candidates ∪ d;
14:   else
15:      drop d from candidates if present;
16:   end if
17:   min-k := min{worstscore(d')|d' ∈ top-k};
18:   if (scanDepth % b == 0) then
19:      for all d' ∈ candidates do
20:         update bestscore(d') using current high_i;
21:         if (bestscore(d') ≤ min-k
               or P[bestscore(d') > min-k] < ε) then
22:            drop d' from candidates;
23:         end if
24:      end for
25:   end if
26:   if (candidates = ∅
         or max{bestscore(d')|d' ∈ candidates} ≤ min-k) then
27:      return top-k;
28:   end if
29: end for
```

4. DYNAMIC QUERY EXPANSION

4.1 Thesaurus-based Expansion Terms

We generate potential expansion terms for queries using a thesaurus based on WordNet [15]. WordNet concepts (i.e., word senses) and their hypernymy/hyponymy relationships are represented as a graph. The edges in the graph are weighted by precomputing statistical correlation measures on large text corpora (e.g., the various TREC corpora). More specifically, we compute the Dice coefficient for each pair of adjacent nodes, counting a document as a co-occurrence of the two concepts, if it contains at least one term or phrase denoting a concept from each of the two sets of synonyms (synsets) that WordNet provides for the concepts. For non-adjacent nodes the similarity weight is computed on demand by multiplying edge weights along the best connecting path, using a variant of Dijkstra's shortest path algorithm.

A query term t is mapped onto a WordNet concept c by comparing the textual context of the query term (i.e., the description of the query topic or the summaries of the top-10 results of the original query) against the synsets and glosses (i.e., short descriptions) of concept c and its hyponyms (and optionally siblings in the graph). This comparison computes a cosine similarity between the two contexts $con(t)$ and $con(c)$ for each potential target concept c whose synset contains t. The mapping uses a simple form of word sense disambiguation by choosing the concept with the highest cosine similarity. Here $con(t)$ and $con(c)$ are basically bags of words, but we also extract noun phrases (e.g., "fiber optics" or "crime organization") and include them as features in the similarity computation by forming 2-grams and 3-grams and looking them up in WordNet's synsets. Once we have mapped a query term t onto its most likely concept c, we generate a set $exp(t)$ of potential expansion terms. Each $t_j \in exp(t)$ has a weight $sim(t, t_j)$ set to the Dice coefficient for adjacent concepts c and c_j (or 1 for the special case of t and t_j being synonyms) or an aggregation of edge weights on the shortest connecting path between c and c_j as described above.

4.2 Incremental Merge Algorithm

At query run-time, for a query with terms $t_1 \ldots t_m$ we first look up the potential expansion terms $t_{ij} \in exp(t_i)$ for each t_i with $sim(t_i, t_{ij}) > \theta$, where θ is a fine-tuning threshold for limiting $exp(t_i)$. Note that θ merely defines an upper bound for the number of expansion terms; in contrast to a static

[1] This assumes independence between occurrences of different terms. The assumption is unlikely to hold on real-life data, but it is often postulated for tractability and seems to go a long way in terms of accuracy and resulting performance gains.

expansion, our algorithm does not necessarily exhaust the full set, such that θ could even be set to 0 in the incremental expansion setup.

The top-k algorithm is extended such that it now merges multiple index lists L_{ij} in descending order of the combined score that results from the local score $s_{ij}(d)$ of an expansion term t_{ij} in a document d and the thesaurus-based similarity $sim(t_i, t_{ij})$. Moreover, to reduce the danger of topic drift, we use the following modified score aggregation for a query $t_1 \ldots t_m$ that counts only the *best match* for any one of the expansion terms per document:

$$score(d) := \sum_{i=1..m} \max_{t_{ij} \in exp(t_i)} \{sim(t_i, t_{ij}) \cdot s_{ij}(d)\}$$

with analogous formulations for the $worstscore(d)$ and $bestscore(d)$ bounds.

The Incremental Merge algorithm provides sorted access to a "virtual" index list that results from merging a set of stored index lists in descending order of the combined $sim(t_i, t_{ij}) \cdot s_{ij}(d)$ values. The index lists for the expansion terms for a given query term are merged on demand (hence, incrementally) until the *min-k* threshold termination is reached, by using the following scheduling procedure for the index scan steps: the next scan step is always performed on the list L_{ij} with the currently highest value of $sim(t_i, t_{ij}) \cdot high_{ij}$, where $high_{ij}$ is the last score seen in the index scan (i.e., the upper bound for the unvisited part of the list). This procedure guarantees that index entries are consumed in exactly the right order of descending $sim(t_i, t_{ij}) \cdot s_{ij}(d)$ products. Index lists for potential expansion terms are opened and added to the set of active expansions $activeExp(t_i)$ for term t_i, only if they are beneficial for identifying a top-k candidate. Often the scan depth on these lists is much smaller than on the index lists for the original query terms, and for many terms in $exp(t_i)$ the index scans do not have to be opened at all, i.e., $activeExp(t_i) \subseteq exp(t_i)$.

The precomputed $sim(t_i, t_{ij})$ values are stored in a meta-index table which is fairly small and typically fits into main memory (e.g., all WordNet nouns together with their initial *high* scores in the inverted lists). The $sim(t_i, t_i)$ is defined to yield the maximum similarity value of 1, such that the Incremental Merge scans are initialized on the index lists for the original query conditions t_i first. The meta-index also contains the maximum possible scores of all expansion candidates, i.e., the initial $high_{ij}$ values at the start of the query execution. This way, we are in a position to open the scans on the expansion-term index lists *as late as possible*, namely, when we actually want to fetch the first index entry from such a list. Fig. 2 gives pseudo code for the Incremental Merge algorithm. Fig. 1 illustrates the merged index scans for an expansion $exp(t)$.

When we want to employ probabilistic pruning based on score predictors, we face an additional difficulty with query expansion using the Incremental Merge technique. Before computing the convolution over the still unknown scores for a subset of the original query's terms, we need to consider the possible expansions for a query term. For term t_i with a random variable S_i capturing the score distribution in the not yet visited part, we are interested in the probability $P[S_i \leq x | S_i \leq high_i]$. With expansion into $exp(t_i) = \{t_{i1}, ..., t_{ip}\}$, this becomes $P[max\{sim_{i1} \cdot S_{i1}, \ldots, sim_{ip} \cdot S_{ip}\} \leq x \mid S_{i1} \leq high_{i1} \wedge \cdots \wedge S_{ip} \leq high_{ip}]$, where sim_{ij} is shorthand for $sim(t_i, t_{ij})$. As the individual factors of this product are captured by the precomputed histograms

Algorithm 2 IncrementalMerge(query $q = (t_1, \ldots, t_m)$, meta-index with $exp(t_1), \ldots, exp(t_m)$ and $sim(t_i, t_{ij})$):

```
1: top-k := ∅; candidates := ∅; min-k := 0;
2: //initialize active expansions
3: for all t_i with i=1..m do
4:    activeExp(t_i) := {t_i};
5:    nextExp(t_i) := t_ij ∈ (exp(t_i) − t_i) with max_{t_ij}{sim(t_i, t_ij)};
6:    nextExpHigh(t_i) := sim(t_i, nextExp(t_i)) · high_{i,nextExp(t_i)};
7: end for
8: //round-robin schedule for sorted accesses
9: for all i=1..m do
10:   best := t_ij ∈ activeExp(t_i)
11:        with max_{t_ij}{sim(t_i, t_ij) · high_ij};
12:   if (sim(t_i, best) · high_best < nextExpHigh(t_i)) then
13:      best := nextExp(t_i);
14:      activeExp(t_i) := activeExp(t_i) ∪ {best};
15:      nextExp(t_i) := t_ij ∈ (exp(t_i) − best)
16:         with max_{t_ij}{sim(t_i, t_ij) · high_ij};
17:      nextExpHigh(t_i) :=
18:         sim(t_i, nextExp(t_i)) · high_{i,nextExp(t_i)};
19:   end if
20:   //next sorted access to L_best
21:   s_i(d) := sim(t_i, best) · L_best.getNext();
22:   //continue as in Prob-k algorithm
23: end for
```

per index list (with the constant $sim(t_i, t_{ij})$ coefficients simply resulting in a proportionally adjusted distribution), we can easily derive a new combined histogram for the max-distribution. This *meta histogram* construction is performed prior to query execution, and it is updated whenever a new scan on another index list is opened with linear costs in the number of histogram cells and the current number of active query expansions. Then this meta histogram is fed as input into the convolution with other (meta) histograms for the original query terms (or their expansion sets). In our experiments the overhead for this dynamic histogram construction was negligible compared to the costs saved in physical disk accesses.

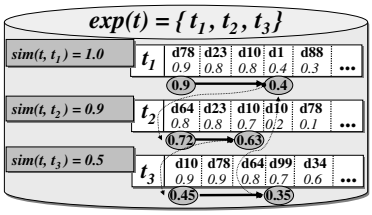

Figure 1: Example schedule for Incremental Merge

5. PHRASE MATCHING

For phrase matching and computing phrase scores, we cannot directly apply the algorithms described so far, because it is rarely feasible to identify potentially query-relevant phrases in advance and create inverted index lists for entire phrases. We store term positions in a separate index, not in the inverted index lists to keep the inverted index as compact as possible and minimize disk I/O when scanning these lists. Thus, testing the adjacency condition (or other kinds of proximity condition and computing an additional proximity factor for the overall score contribution) requires random I/O. We therefore treat the adjacency tests as expensive predicates in the sense of [9] and postpone their evaluation as much as possible.

In our approach we assume that the score for a phrase match of a phrase $t_1 \ldots t_p$ is the sum of the scores for the individual words, possibly normalized by the phrase length.

The key idea to minimizing the random I/Os for adjacency tests is to eliminate many candidates based on their upper score bounds (or estimated quantile of their scores) early before checking for adjacency. Consider a phrase $t_1 \ldots t_p$ and suppose we have seen the same document d in the index lists for a subset of the phrase's words, say in the lists for $t_1 \ldots t_j$ $(j < p)$. At this point we know the $bestscore'(d)$ with regard to this phrase match alone (which is part of a broader query), and we can compare this $bestscore'(d)$ or the total $bestscore(d)$ for the complete query against the $worstscore(d')$ of other candidates d' that have been collected so far. In many cases we can eliminate d from the candidate pool without ever performing the adjacency test. Even if we have seen d in all p index lists for the phrase's words, we may still consider postponing the adjacency test further, as we gain more information about the total $bestscore(d)$ for the entire query (not just the phrase) and the evolving *min-k* threshold of the current top-k documents.

This consideration is implemented using a top-k algorithm for both the phrase matching and the entire query, leading to a notion of *nested top-k operators* that dynamically combine the index lists for each of several phrases of the full query. As indicated by the example expansion Fig. 2, this method allows a very fine grained modeling of semantic similarities directly in the query processing. The algorithm works by running outer and inner top-k operators in separate threads, with inner operators periodically reporting their currently best candidates along with the corresponding $[worstscore_i(d),\ bestscore_i(d)]$ intervals. Then the aggregated score interval of a candidate d at each operator simply becomes the *sum* of the score interval boundaries propagated by each of the subordinate operators $[\sum_{i=1..m} worstscore_i(d),\ \sum_{i=1..m} bestscore_i(d)]$ for a top-k join or the maximum of the respective interval boundaries $[\max_{i=1..m} worstscore_i(d),\ \max_{i=1..m} bestscore_i(d)]$ for an Incremental Merge join which remains a monotonous score aggregation at any level of the operator tree. At the synchronization points, the outer operator integrates the reported information into its own candidate bookkeeping and considers pruning candidates from the outer priority queue. This technique also allows a lazy scheduling of random I/Os for phrase tests by the top-level operator only which can further decrease the amount of phrase tests by an order of magnitude for large phrase expansions.

6. EXPERIMENTAL EVALUATION

The presented algorithms are implemented in a Java prototype using JDK 1.4. Inverted index lists are stored as tables with appropriate B$^+$-tree indexes in an Oracle 10g database; in addition, a term-to-position index is kept in a second table for phrase matching. The top-k engine accesses the index lists via JDBC, with a large prefetching buffer. Index scans are multi-threaded, with periodic thread synchronization and queue updates after every $b = 500$ index scan steps. All experiments were run on an Intel Dual XEON with 4 GB RAM and a large RAID-5 disk array.

6.1 Setup & Datasets

We performed experiments with two different TREC data sets and ad-hoc query tasks: (1) The distinct set of 50 queries of the TREC 2004 Robust track marked as *hard* (explicitly identified by TREC) on the Aquaint news corpus,

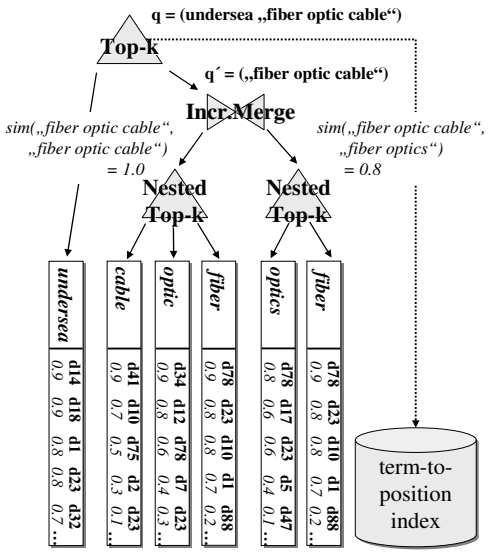

Figure 2: Phrase matching with multiple nested top-k operators

consisting of 528,155 text documents and a raw data size of 1.9 GB and yielding about 86 million distinct (term, docid, score) tuples; and (2) the 50 queries of the TREC 2004 Terabyte track on recently crawled web documents of the .gov Internet domain, consisting of 25,205,179 documents and a raw data size of 426 GB and yielding about 2.9 billion distinct (term, docid, score) tuples.

For both collections we used a standard Okapi BM25 [31, 32] model with the exact parameters k_1 and b set to 1.2 and 0.75, respectively. Query expansion was based on WordNet concepts and the statistically quantified hypernym/hyponym relations in addition to the concept synsets (see Sec. 4.1). We used both the benchmark query titles, which are merely 2 to 5 keywords, and the descriptions, which are a few sentences describing the query topic. We mostly focus on results for the 50 hard queries of the Robust track data as these have become the gold standard for query expansion; we will discuss results for the Terabyte corpus only in Sec. 6.4 on scalability. We compared both efficiency and effectiveness of the following methods:

a) a *baseline* method without query expansion, with the option of probabilistic candidate pruning and variation of the control parameter ϵ (with $\epsilon = 0.0$ being the conservative case where all speculative and approximative techniques are disabled),

b) a *static expansion* method, where we generate a large disjunctive query by adding all expansion terms with similarity to at least one of the original query terms above a threshold θ (with $\theta = 1.0$ being the special case where only synonyms are added), and

c) the *Incremental Merge* method for dynamic on-demand expansion, with an upper bound on the number of expansion terms controlled by θ.

With static expansion the score aggregation function was the summation of scores for the whole expansion; with Incremental Merge we used the score aggregation function introduced in Sec. 4.2. For both expansion methods we varied the control parameter ϵ for probabilistic candidate pruning. In addition to these methods under comparison, we also indicate the number of relevant entries of index lists for a

DBMS-style *Join&Sort* query processing, where all inverted index lists that are relevant to a query are fully scanned (however, disregarding random I/Os for phrase matching); this merely serves as a reference point. We measured the following metrics for assessing efficiency and effectiveness:

- the *total number of sequential accesses* to inverted index lists for all benchmark queries, i.e., the number of (docid, score) tuples read from the inverted lists,
- the *total number of random accesses* to the index lists and the position index (for phrase matching),
- the *total CPU time* for all benchmark queries,
- the *maximum memory* consumption during the benchmark run (for the candidate pool, probabilistic predictors, etc.),
- the *macro average precision@10* for the top-10 result of each query, using the official TREC 2004 relevance assessments,
- the *uninterpolated mean average precision* (MAP) for the top-1000 of each query (using separate runs to obtain the top-1000 results), and
- the *macro average relative precision* $rPrec(R_1, R_2) := \frac{|R_1 \cap R_2|}{\max\{|R_1|,|R_2|\}}$ for the top-10 of each query compared to the conservative algorithm with $\epsilon = 0$ (a relative overlap measure for the probabilistic pruning).

6.2 Baseline Runs

The first part of Tab. 1 shows the results for the baseline run, i.e., without query expansion. The maximum query dimensionality $Max(m)$ was 4, the average query length $Avg(m)$ was 2.5 terms; there was no consideration of phrases. We see that top-k query processing is generally much more efficient than the Join&Sort technique could possibly be, in terms of inverted index lists accesses. The rows for the top-k baseline method differ in their settings for the ϵ control parameter. Allowing a modest extent of probabilistic pruning led to noticeable performance gains, while it hardly affects effectiveness. This makes the Prob-k method an excellent choice for a system where high precision at the top ranks is required and efficiency is crucial.

6.3 Expansion Strategies

The second part of Tab. 1 shows the efficiency and effectiveness results for the 50 hard queries of the TREC Robust track, comparing the Incremental Merge method with static query expansion. A fixed expansion technique using only synonyms and first-order hyponyms of noun-phrases from titles and descriptions already produced fairly high-dimensional queries, with up to 118 terms (many of them marked as phrases); the average query size was 35 terms.

Compared to the baseline without query expansion, all expansion techniques significantly improved the result quality in terms of precision@10 and MAP. The quality is still below the very best TREC runs [20, 21] which achieved about 0.37 precision@10, but the results are decent. Recall that our basis for query expansion, WordNet, is certainly not the most suitable choice for ad-hoc query expansion (at least not unless combined with other sources and techniques), and the emphasis of this paper is on efficiency with good (but not hand-tuned) effectiveness. The best result quality in our experiment, 0.31 precision@10, was achieved by the Incremental Merge technique with ϵ set to 0.0. Probabilistic pruning with $\epsilon = 0.1$ reduced the number of sorted accesses by about 25 percent, but in terms of run-time (in CPU seconds) it gained a factor of 2 as it also incurred fewer random accesses (mostly for phrase matching) and lower memory overhead. For the static expansion technique, the cost savings by the probabilistic pruning were much higher, more than a factor of 4 in all major efficiency metrics, but this came at the expense of a significant loss in query result quality. In terms of run-time, however, both Incremental Merge and static expansion performed almost equally well when probabilistic pruning was used. The absolute performance numbers of less than 2 seconds per query, with an academic prototype in Java and the high overhead of JDBC sessions, are very encouraging. It is particularly remarkable that for the Incremental Merge method, the probabilistic pruning affected the effectiveness only to a fairly moderate degree, reducing precision@10 from 0.310 to 0.298. This seems to be a very low price for a speed-up factor of 2.

Fig. 3 and 4 show how the effectiveness and efficiency measures change as the control parameter ϵ is varied from 0.0 and 0.1 towards higher, more aggressive values (with the extreme case 1.0 corresponding to a fixed amount of index-scan work, looking only at the $b = 500$ highest-score entries of each index list). We see that the relative precision of most variants drops linearly with ϵ, which is just the expected behavior (see [35]), but in terms of objective result quality, as measured by the TREC relevance assessments, the pruning technique performed much better: both precision@10 and MAP decrease only very moderately with increasing ϵ. For example, for $\epsilon = 0.5$ the best Incremental Merge method could still achieve a precision@10 of about 0.27 while its run-time cost, in terms of sorted accesses, was reduced by a factor of more than 3. In Fig. 3 and 4 the curves for static expansion with query titles and descriptions reveal that the score predictor degenerates for very high-dimensional disjunctive queries because of neglecting feature correlations. This led to the sudden drop in the number of sorted accesses already for ϵ being as low as 0.1, but this came at the expense of a significant loss in retrieval quality. This phenomenon did not occur for the Incremental Merge method, where expansions are grouped into multiple nested top-k operators, such that the maximum number of query dimensions at the top level was only 22 compared to 118 for the static expansion (including phrases).

Fig. 5 and 6 show the efficiency and effectiveness results as a function of varying the θ parameter, i.e., the aggressiveness of generating candidate terms and phrases for query expansion. Tab. 2 gives detailed figures for various combinations of θ and ϵ values. The charts demonstrate the robustness and self-tuning of the Incremental Merge method: even with very low θ, meaning very aggressive expansion, both precision@10 and MAP values stayed at a very good level. The execution cost did significantly increase, however, but this is not surprising when you consider that the expansion candidate sets for some queries had up to 876 terms or phrases – quite a stress test for any search engine. In combination with moderate probabilistic pruning, the Incremental Merge method was still able to answer all 50 queries in a few minutes, less than 4 seconds per query.

6.4 Scalability

The evaluation on the Terabyte corpus and queries served as a proof of scalability for the developed top-k query processing and expansion methods. The third part of Tab. 1

	ϵ	Max(m)	Avg(m)	# Seq	# Rand	CPU Sec	Memory	P@10	MAP	rPrec
Robust Baseline Runs										
Join&Sort (title only)		4	2.5	2,305,637						
Title only	0.0	4	2.5	1,439,815	0	9.4	431 KB	0.252	0.092	1.000
	0.1	4	2.5	1,339,756	0	9.3	432 KB	0.248	0.091	0.934
Robust Fixed Expansions										
Join&Sort (static expansion)		118	35,38	20,582,764						
Static expansion	0.0	118	35.38	18,258,834	210,531	245.0	37,355 KB	0.286	0.105	1.000
	0.1	118	35.38	3,622,686	49,783	79.6	5,895 KB	0.238	0.086	0.541
Incr. merge	0.0	118	35.38	7,908,666	53,050	159.1	17,393 KB	0.310	0.118	1.000
	0.1	118	35.38	5,908,017	48,622	79.4	13,424 KB	0.298	0.110	0.786
Terabyte Fixed Expansions										
Join&Sort (static expansion)		119	33.54	581,307,315						
Static expansion	0.0	119	33.54	360,811,608	973,188	18,090.1	783,273 KB	0.234	0.079	1.000
	0.1	119	33.54	63,028,142	120,180	6,785.5	101,333 KB	0.216	0.071	0.631
Incr. merge	0.0	119	33.54	87,327,339	211,981	10,734.3	575,440 KB	0.272	0.102	1.000
	0.1	119	33.54	64,548,694	147,238	7,966.5	540,845 KB	0.269	0.099	0.707

Table 1: Baseline and fixed expansions for the 50 hard Robust and the 50 Terabyte queries

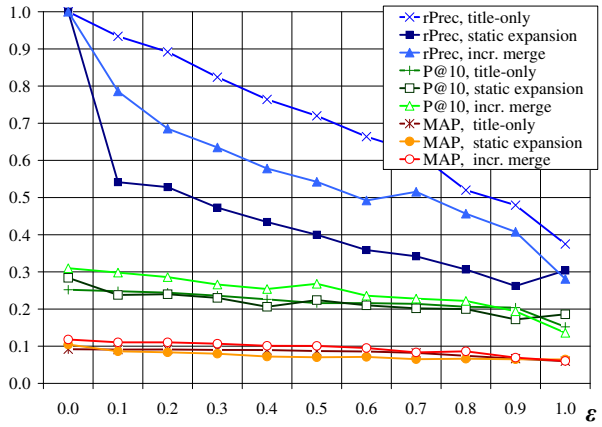

Figure 3: Precision as a function of ϵ

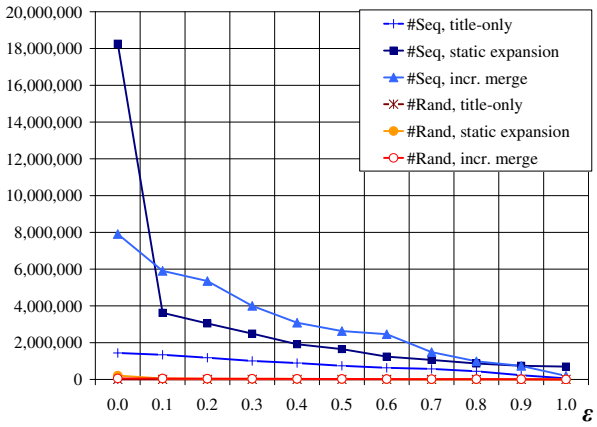

Figure 4: Efficiency as a function of ϵ

shows the results of some of our runs, using the same fixed expansion technique as in Sect. 6.3 for the Robust track. The performance gap for static expansions without vs. the one with probabilistic pruning indicates the same overly aggressive behavior of the score predictor as for the high dimensional queries in the Robust setup – however, again at the expense of retrieval quality. The overall efficiency gain in terms of access rates for the Incremental Merge method compared to the static expansion without probabilistic pruning is even more impressive by a factor of more than 4 and a factor of 8 compared to full scans, respectively, while achieving higher precision@10 and MAP values. This makes the Incremental Merge approach the method of choice in terms of both retrieval robustness and efficiency.

7. CONCLUSION

This paper has presented a suite of novel techniques for efficient query expansion and tested them on the TREC Robust and Terabyte benchmarks. The Incremental Merge technique clearly outperformed the traditional method of static expansion, and proved that it can achieve decent query-result quality while exhibiting very good execution cost and run-time behavior. Especially in combination with probabilistic score prediction and candidate pruning, it is a very efficient and effective, scalable method that could be of interest for industrial-strength search engines. Our emphasis has been on efficiency, so we used only a relatively simple basis for generating expansion terms and phrases. Our future work may consider combining our methods with more advanced techniques for expansion, using, for example, document summaries from Google top-10 snippets and query associations from query logs. A second line of ongoing and future work is to study more sophisticated scheduling strategies for random accesses to inverted index lists and position indexes. Our current strategies all have a greedy-heuristics flavor; a more advanced approach could be to drive the scheduling of sorted and random accesses by considering score distributions and an explicit cost model.

8. REFERENCES

[1] V.N. Anh, O. de Kretser, A. Moffat: Vector-Space Ranking with Effective Early Termination, SIGIR 2001

[2] V.N. Anh, A. Moffat: Impact Transformation: Effective and Efficient Web Retrieval, SIGIR 2002

[3] B. Billerbeck, F. Scholer, H.E. Williams, J. Zobel: Query expansion using associated queries, CIKM 2003

[4] B. Billerbeck, J. Zobel: Questioning Query Expansion: An Examination of Behaviour and Parameters, Australian Database Conference (ADC), 2004

[5] B. Billerbeck, J. Zobel: Techniques for Efficient Query Expansion, SPIRE 2004

[6] A.Z. Broder et al.: Efficient Query Evaluation using a Two-Level Retrieval Process, CIKM 2003

	θ	Max(m)	Avg(m)	#Seq	#Rand	CPU Sec	Memory	P@10	MAP	rPrec
Join&Sort	1.00	36	13.32	7,655,462						
	0.30	59	22.12	16,157,145						
	0.10	102	38.90	30,288,748						
	0.01	876	230.82	243,394,509						
Static expansion	1.00	36	13.32	5,427,347	169,635	38.5	8,987 KB	0.274	0.107	1.000
$\epsilon = 0.0$	0.30	59	22.12	10,586,175	555,176	156.6	29,913 KB	0.286	0.111	1.000
	0.10	102	38.90	15,541,754	1,950,718	230.8	60,195 KB	0.272	0.105	1.000
	0.01	876	230.82	47,169,998	4,575,223	679.2	755,792 KB	0.256	0.098	1.000
Incr. merge	1.00	36	13.32	4,850,129	60,739	33.6	6,802 KB	0.278	0.122	1.000
$\epsilon = 0.0$	0.30	59	22.12	7,575,647	65,523	56.8	13,867 KB	0.276	0.113	1.000
	0.10	102	38.90	10,889,717	77,261	100.1	25,628 KB	0.272	0.104	1.000
	0.01	876	230.82	31,023,932	102,669	407.6	68,592 KB	0.261	0.096	1.000
Incr. merge	1.00	36	13.32	3,668,119	30,759	26.2	3,257 KB	0.280	0.117	0.645
$\epsilon = 0.1$	0.30	59	22.12	5,671,493	34,895	43.8	7,931 KB	0.276	0.110	0.649
	0.10	102	38.90	7,615,389	38,641	78.3	24,454 KB	0.275	0.102	0.670
	0.01	876	230.82	16,748,953	73,153	189.4	46,456 KB	0.278	0.102	0.668

Table 2: Various θ expansions for the 50 hard Robust queries

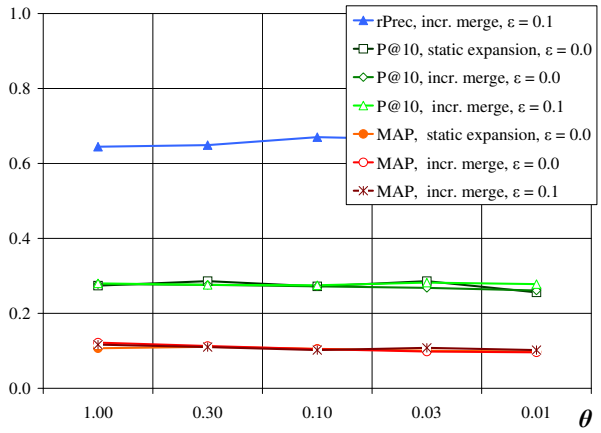

Figure 5: Precision as a function of θ

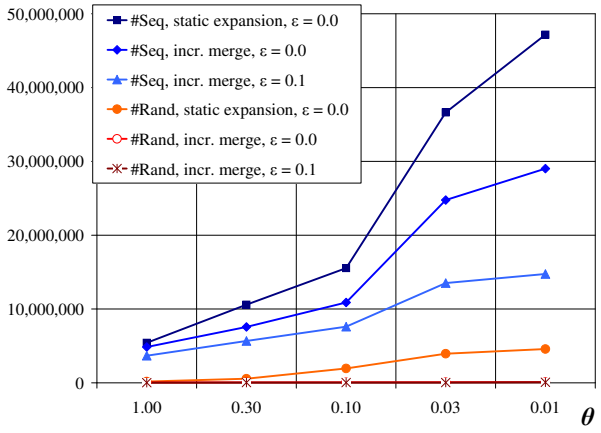

Figure 6: Efficiency as a function of θ

[7] C. Buckley, A. F. Lewit: Optimization of Inverted Vector Searches, SIGIR 1985

[8] C. Buckley, G. Salton, J. Allan: The Effect of Adding Relevance Information in a Relevance Feedback Environment, SIGIR 1994

[9] K.C.-C. Chang, S.-W. Hwang: Minimal Probing: Supporting Expensive Predicates for Top-k Queries, SIGMOD 2002

[10] S. Chaudhuri, L. Gravano, A. Marian: Optimizing Top-K Selection Queries over Multimedia Repositories, TKDE 16(8), 2004

[11] S. Cronen-Townsend, Y. Zhou, W.B. Croft: A Framework for Selective Query Expansion, CIKM 2004

[12] A.P. de Vries et al.: Efficient k-NN Search on Vertically Decomposed Data, SIGMOD 2002

[13] R. Fagin: Combining Fuzzy Information: an Overview, ACM SIGMOD Record 31(2), 2002

[14] R. Fagin et al.: Optimal aggregation algorithms for middleware, J. Comput. Syst. Sci. 66(4), 2003

[15] C. Fellbaum (Ed.): WordNet: An Electronic Lexical Database, MIT Press, 1998

[16] U. Güntzer, W.-T. Balke, W. Kießling: Optimizing Multi-Feature Queries for Image Databases, VLDB 2000

[17] U. Güntzer, W.-T. Balke, W. Kießling: Towards Efficient Multi-Feature Queries in Heterogeneous Environments, ITCC 2001

[18] C.-K. Huang, L.-F. Chien, Y.-J. Oyang: Relevant term suggestion in interactive web search based on contextual information in query session logs. JASIST 54(7), 2003

[19] I. F. Ilyas, W. G. Aref, A. K. Elmagarmid: Supporting top-k join queries in relational databases, VLDB J. 13(3), 2004

[20] K.L. Kwok et al.: TREC2004 Robust Track Experiments using PIRCS, TREC 2004

[21] S. Liu, F. Liu, C. Yu, W. Meng: An Effective Approach to Document Retrieval via Utilizing WordNet and Recognizing Phrases, SIGIR 2004

[22] X. Long, T. Suel: Optimized Query Execution in Large Search Engines with Global Page Ordering, VLDB 2003

[23] A. Marian, N. Bruno, L. Gravano: Evaluating Top-k Queries over Web-Accessible Databases. ACM TODS 29(2), 2004

[24] M. Mitra, A. Singhal, C. Buckley: Improving Automatic Query Expansion, SIGIR 1998.

[25] A. Moffat, J. Zobel: Self-Indexing Inverted Files for Fast Text Retrieval. ACM TOIS 14(4), 1996

[26] A. Natsev et al.: Supporting Incremental Join Queries on Ranked Inputs, VLDB 2001

[27] S. Nepal, M. V. Ramakrishna: Query Processing Issues in Image (Multimedia) Databases, ICDE 1999

[28] M. Persin, J. Zobel, R. Sacks-Davis: Filtered Document Retrieval with Frequency-Sorted Indexes. JASIS 47(10), 1996

[29] U. Pfeifer, N. Fuhr: Efficient Processing of Vague Queries using a Data Stream Approach, SIGIR 1995

[30] Y. Qiu, H.P. Frei: Concept Based Query Expansion, SIGIR 1993

[31] S.E. Robertson, S. Walker: Some Simple Effective Approximations to the 2-Poisson Model for Probabilistic Weighted Retrieval, SIGIR 1994

[32] S.E. Robertson, H. Zaragoza, M. Taylor: Simple BM25 extension to multiple weighted fields, CIKM 2004

[33] F. Scholer, H.E. Williams, A. Turpin: Query Association Surrogates for Web Search, JASIST 55(7), 2004

[34] C. Stokoe, M.P. Oakes, J. Tait: Word Sense Disambiguation in Information Retrieval Revisited, SIGIR 2003

[35] M. Theobald, G. Weikum, R. Schenkel: Top-k Query Evaluation with Probabilistic Guarantees, VLDB 2004

[36] E.M. Voorhees: Query Expansion using Lexical-Semantic Relations, SIGIR 1994

[37] J. Xu, W.B. Croft: Query Expansion Using Local and Global Document Analysis, SIGIR 1996

[38] C.T. Yu, P. Sharma, W. Meng, Y. Qin: Database selection for processing k nearest neighbors queries in distributed environments, JCDL 2001