IO-Top-k at TREC 2006: Terabyte Track

Holger Bast Debapriyo Majumdar Ralf Schenkel Martin Theobald Gerhard Weikum

Max-Planck-Institut für Informatik, Saarbrücken, Germany
{bast,deb,schenkel,mtb,weikum}@mpi-inf.mpg.de

ABSTRACT

This paper describes the setup and results of our contribution to the TREC 2006 Terabyte Track. Our implementation was based on the algorithms proposed in [1] *"IO-Top-k: Index-Access Optimized Top-K Query Processing, VLDB'06"*, with a main focus on the efficiency track.

1. INTRODUCTION

IO-Top-k [1] extends the family of *threshold algorithms (TA)* [3, 4, 8] with a suite of new strategies. To retrieve the best-scoring (so-called top-k) answers to a multi-keyword query under a monotonic aggregation of per-keyword scores, TA-style algorithms perform index scans (so-called sorted accesses) over precomputed index lists, one for each keyword in the query, which are sorted in descending order of per-keyword scores. The key point of TA is that it aggregates scores on the fly, thus computes a lower bound for the total score of the current rank-k result document and an upper bound for the total scores of all other candidate documents, and is thus often able to terminate the index scans long before it reaches the bottom of the index lists, namely, when the lower bound for the rank-k result, the *threshold*, is at least as high as the upper bound for all other candidates. Additional, carefully selected random accesses to reveal the score of a candidate document in a list where it has not yet been seen so far can further speed up the computation. The goal of such algorithms is to minimize *the sum of the access costs*, assuming a fixed cost c_S for each sorted access and a fixed cost c_R for each random access. In a realistic scenario, random accesses are a factor of 50 to 50,000 more expensive than sorted accesses. We aim at accelerating queries and, at the same time, limit or even aim to reducing the memory consumption for candidate queues and other auxiliary data structures.

For our participation in TREC 2006, we selected two strategies from the suite of algorithms presented in [1]:

- A scheduling strategy for random accesses that postpones all random accesses to the end of the execution, switching from scans to random accesses when the estimated cost for them is the same.
- A heuristics for early termination that scans only a configurable fraction of the lists, regardless of the score bounds.

2. COMPUTATIONAL MODEL AND SCORING

We associate with each document-term pair a numeric *score* that reflects the "goodness" or relevance of the data item with regard to the term. As effectiveness was not in the focus of our experiments, we chose the well-known probabilistic Okapi BM25 score derived from term frequencies (TF) and inverse document frequencies (IDF) [9]. We boost the frequency of terms within important tags (like `title`, `h1`, or `caption`) by an additional, tag-specific weight. Denoting the score of document d_j for the ith dimension by s_{ij}, we get

$$s_{ij} = \frac{(k_1+1)\cdot tf_i(d_j)}{K + tf_i(d_j)} \cdot \log \frac{N - df_i + 0.5}{df_i + 0.5}$$

where $tf_i(d_j)$ is the term frequency of term i in document d_j, df_i is the document frequency of term i, and

$$K = k_1 \cdot \left((1-b) + b\frac{length(d_j)}{avg(length(d))}\right)$$

For our experiments, we chose $k_1 = 1.2$ and $b = 0.75$. All scores are normalized to the interval $[0, 1]$, with 1 being the best possible score.

A top-k query asks for those k documents with the highest score sum. Note that such a document must not necessarily contain all query words, because a document containing just some of the query words but with a high score each can have a larger score sum than a document which contains all query words, but with only a relatively low score each.

3. INVERTED BLOCK-INDEX

The documents that contain specific terms and their corresponding scores are precomputed and stored in inverted index lists L_i ($i = 1..M$). There is one such index list per term. The entries in a list are `<docID, score>` pairs. The lists may be very long (millions of entries) and reside on disk, with a B^+-tree or similar data structure for efficiently locating the keys of the lists (i.e., the attribute values or terms). We partition each index list into blocks and use score-descending order among blocks but keep the index entries within each block in `docID` order. This special ordering, which is halfway between an ordering by score and an ordering by doc id, is key to efficiently manage the substantial bookkeeping required in TA-style query processing.

The block size is a configuration parameter that is chosen in a way that balances disk seek time and transfer rate; a typical block size would be 32,768.

4. QUERY PROCESSING

Our query processing model is based on the NRA and CA variants of the TA family of algorithms [3]. An m-dimensional top-k query (with m search conditions) is primarily processed by scanning the corresponding m index lists in descending score orders in an interleaved, round-robin manner (and by making judicious random accesses to look up index entries of specific documents). Without loss of generality, we assume that these are the index lists numbered L_1 through L_m.

When scanning the m index lists, the query processor collects candidates for the query result and maintains them in two priority queues, one for the *current top-k items* and another one for *all other candidates* that could still make it into the final top-k. For simpler presentation, we assume that the score aggregation function is simple summation (but it is easy to extend this to other monotonic functions). The query processor maintains the following state information:

- the current cursor position pos_i for each list L_i,
- the score values $high_i$ at the current cursor positions, which serve as upper bounds for the unknown scores in the lists' tails,
- a set of current top-k items, d_1 through d_k (renumbered to reflect their current ranks) and a set of data items d_j ($j = k+1..k+q$) in the current candidate queue Q, each with
  - a set of evaluated dimensions $E(d_j)$ in which d_j has already been seen during the scans or by random lookups,
  - a set of remainder dimensions $\bar{E}(d_j)$ for which the score of d_j is still unknown,
  - a lower bound $worstscore(d_j)$ for the total score of d_j which is the sum of the scores from $E(d_j)$,
  - an upper bound $bestscore(d_j)$ for the total score of d_j which is equal to
$$worstscore(d_j) + \sum_{\nu \in \bar{E}(d_j)} high_\nu$$
(and not actually stored but rather computed from $worstscore(d_j)$ and the current $high_\nu$ values whenever needed).

In addition, the following information is derived at each step:

- the minimum worstscore *min-k* of the current top-k docs, which serves as the stopping threshold,
- the bestscore that any currently unseen document can get, which is computed as the sum of the current $high_i$ values, and
- and for each candidate, a score deficit $\delta_j = min\text{-}k - worstscore(d_j)$ that d_j would have to reach in order to qualify for the current top-k.

The top-k queue is sorted by worstscore values, and the candidate queue is sorted by descending bestscore values. Ties among scores may be broken by using the concatenation of `<score, docID>` for sorting. The invariant that separates the two is that the rank-k worstscore of the top-k queue is at least as high as the best worstscore in the candidate queue. The algorithm can safely terminate, yielding the correct top-k results, when the maximum bestscore of the candidate queue is not larger than the rank-k worstscore of the current top-k, i.e., when

$$\min_{d \in top\text{-}k} \{worstscore(d)\} =: min\text{-}k \geq \max_{c \in Q} \{bestscore(c)\}$$

More generally, whenever a candidate in the queue Q has a bestscore that is not higher than *min-k*, this candidate can be pruned from the queue. Early termination (i.e., the point when the queue becomes empty) is one goal of efficient top-k processing, but early pruning to keep the queue and its memory consumption small is an equally important goal (and is not necessarily implied by early termination). The candidate bookkeeping is illustrated in Fig. 1.

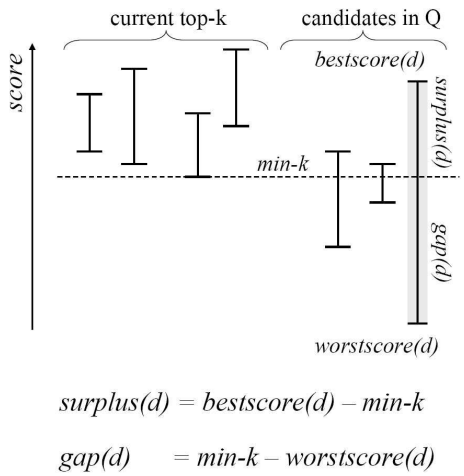

Figure 1: Top-k and candidate bookkeeping.

The state information we have to maintain for each document, from the point it is first encountered to the point where it is surely known that either the document is one of the $top-k$ or that it cannot be, adds a noticeable amount of overhead to the algorithm. This has to be contrasted with a simple full merge (of the lists sorted by document ids), which can compute the full scores document by document, and then determine the top-k items by a (partial) sort. It is not at all obvious, and indeed put forward as an open problem in [3], whether the state maintenance of any of the sophisticated TA-style algorithms can be implemented efficiently enough so that the gains in the abstract cost indeed show in faster running times.

In our first implementation we maintained all state information in a hash data structure; indeed, this is the approach taken in all top-k implementations that we are aware of [11]. However, despite their strong advantage in theoretical cost, none of our sophisticated algorithms could beat the simple full-merge baseline in this implementation. We then switched to the inverted block-index described in Section 3. An essential ingredient of our implementation is to keep the state information *in-place*, i.e., in a contiguous memory segment together with the document id. The process of merging two or more document lists, and updating all state information then has almost optimal locality properties.

The most time-critical step in the merge is the computation of the bestscore, which we do not store explicitly but rather compute from the worstscore and the set of lists in which the documents have been seen so far. We store this *seen* information by a simple m-bit vector, where m is

the number of lists, and for each round precompute all 2^m partial sums of the high-scores $high_i$ of each list (see Section 4). For any document, the bestscore can then be computed from the worstscore by a simple table lookup with the *seen*-bitvector serving as a direct index into that table.

To keep the merges as fast as those of the baseline full-merge, we also do not maintain the set of top-k items as we merge, and not even the *min-k* score. We rather do the merge twice, outputting only the scores in the first round, doing a partial sort of these to obtain the *min-k* score, and then repeat the merge, but this time with an on-the-fly pruning of all documents with a bestscore below that *min-k* score. By these, and a bag of other tricks, we managed to keep the overhead for maintaining the state-information a small fraction of the essential operations of reading and merging blocks of pairs of document ids and score, sorted by document id.

5. SCHEDULING OF RANDOM ACCESSES

Random-access (RA) scheduling is crucial both in the early and the late stages of top-k query processing. In the early stage, it is important to ensure that the *min-k* threshold moves up quickly so as to make the candidate pruning more effective as the scans proceed and collect large amounts of candidates. Later, it is important to avoid that the algorithm cannot terminate merely because of a few pieces of information missing about a few borderline candidates. In [1], we analyzed various strategies for deciding when to issue RAs and for which candidates in which lists; for our experiments, we focused on one of these strategies that was both efficient to compute and resulted in a good performance. Following the literature [2, 6], we refer to score lookups by RA as *probing*. As in [3], we denote by c_S the cost of a sorted access, and by c_R the cost of a random access.

Our scheduling strategy for random accesses, coined *Last-Probing* in [1], does a balanced number of random accesses just as Fagin's CA algorithm, that is, the total cost of the random accesses is about the same as the total cost of all sorted accesses. In CA, this is trivially achieved by doing one random access after each round of $\lceil c_R/c_S \rceil$ sorted accesses. In Last-Probing, we perform random accesses only after the last round, that is, we have a phase of only sorted accesses, followed by a phase of only random accesses.

We do this by estimating, after each round, the number of random accesses that would have to be done if this were the last round of sorted accesses. Two criteria must be met for this round of sorted accesses being the last. First, the estimated number of random accesses must be less than $\lceil c_R/c_S \rceil$ times the number of *all* sorted accesses done up to this point Second, we must have $\sum_{i=1}^{m} high_i \leq min\text{-}k$, since only then we can be sure that we have encountered all the top-k items already. We remark that in all our applications, the second criterion is typically fulfilled long before (that is, after much fewer rounds than) the first criterion. A simple estimate for the number of random lookups that would have to be done if we stopped doing sorted accesses at a certain point, is the number of candidate documents which are then in our queue. When the distribution is very skewed, it is in fact quite a good estimate, because then each document in the queue has a positive but only very tiny probability of becoming one of the top-k items.

When doing the random accesses, it plays a role in which order we process the documents for which we do random lookups. In our algorithm, we first schedule RAs for any items in the current top-k that are not yet completely evaluated, in decreasing order of their worst scores. Then, we schedule RAs for items in the candidate queue, ordered by decreasing bestscore (`Last-Best`). This is similar to CA, which after each round of sorted accesses does a random access for the candidate document with the highest bestscore.

Note that unlike more aggressive pruning strategies proposed in the literature [5, 7, 10] that provide approximate top-k results, our method is non-approximative and achieves major runtime gains with no loss in result precision.

6. EARLY STOPPING HEURISTIC

We used an *early stopping heuristic* for two of our efficiency runs. For these runs, we ignored all the blocks after the first 1/5-th of the blocks in every list (e.g. if there are 14 blocks in a list, we only considered the first 3 blocks). Note that since the blocksize of our inverted block index was as large as 131,072, all the small lists were almost fully scanned. Only for the very long lists, the tails are ignored. As we find from the results (see Section 9.1), this heuristic works quite well in practice.

7. TEST PLATFORM

We parsed the collection on a small cluster of three servers, each with two Intel Xeon processors running Windows at 3 GHz. Our TREC runs were performed on a *single machine* having two 2390 MHz AMD Opteron CPUs and 8 GB of memory. The index files were stored on a local 44 GB SCSI disk with 10,000 rpm rotational speed. The operating system used was Linux.

8. INDEXING

We indexed the collection using Okapi BM25 scoring function with standard parameters after removing stopwords, but without stemming. Prior to computing the BM25 scores, the term frequencies (tfs) of the terms were computed using a weighted sum of term-occurrences, with the weights being one for occurrences in standard text, and between 1.5 and 4 for occurrences inside special HTML tags (see Table 1 for details). The term scores were initially stored in a relational schema of the form `(docID,term,score)` in an Oracle database. After the parsing, the index lists were created from the database and stored in two files, namely one for sorted access and one for random access.

HTML tag	factor
`TITLE, in URL`	4.0
`H1 - H2`	3.0
`H3 - H6, STRONG, B, CAPTION, TH`	2.0
`EM, I, U, DL, OL, UL, A, META`	1.5

Table 1: Term weights for different tags

The inverted block-index for sorted access stored, for each term, a list of pairs of the form `<docID,score>`, together with a two level B-Tree to store the offsets of the lists corresponding to every term. The index for random access stored list of pairs `<termID,score>` for every document and the offsets for the lists were stored in a single array. The total size of our index files on disk was about 35 GB, each index

contributing to little more than 17 GB. At runtime, only the offsets of the lists for random access (about 200 MB) resided in memory, all other data were read from disk. We did not use any caching other than some automatic filesystem caching over which we did not have any control.

9. RUNS AND RESULTS

We submitted four runs for the efficiency task and four runs for the adhoc task. However, this year, our main focus was in the efficiency task.

9.1 Efficiency Task

For all of the efficiency runs, the queries were parsed automatically from the query streams, and all the words present in each query line were taken as keywords. Since stopwords were removed at the time of parsing, such words automatically did not play any role in retrieval.

The four runs submitted for the efficiency task were based on two versions of our algorithm, as follows:

- **mpiiotopk** - computation of exact top-20 documents (as defined in Section 4 and Section 5) for each query. The 100,000 queries were processed sequentially from a single stream. The average running time was 0.152 sec. This run essentially used a single processor.

- **mpiiotopkpar** - computation of exact top-20 documents using the same scheme as in **mpiiotopk**. However, for this run the queries were processed from four streams in parallel. Note that the documents returned by this run were the same as the documents returned by **mpiiotopk**. In spite of processing four streams parallally, this run is only about twice faster (average running time: 0.074 sec) than the previous run, because the machine we used had only two processors.

- **mpiiotopk2** - avoids scanning deep into long lists using the early stopping heuristic as described in Section 6, with all 100,000 queries being processed sequentially from a single stream. The documents returned by this run are not the exact top-20. Using the early stopping heuristic, the average runtime improves by more than 2.5 times (average running time: 0.057 sec) from the exact run.

- **mpiiotopk2p** - same as **mpiiotopk2**, but four query streams were processed in parallel. Again, the parallelization improves the running time only by a factor of two (average running time: 0.028 sec), because the program was run on a machine with two processors. A proper parallelization of the process with at least 4 processors could boost the efficiency of these runs, but we did not have such a setup at the time of performing these experiments.

The documents returned by the first two runs (**mpiiotopk** and **mpiiotopkpar**) are precisely those that any retrieval model using standard BM25 scoring function would return. The precisons of these runs turned out to be decent, namely 0.5110 on average for topics 751-800 and 0.4280 on average for topics 801-850. Interestingly, the precisions of the runs **mpiiotopk2** and **mpiiotopk2p** using the early stopping heuristic were not much worse for topics 751-800 (0.4820) and equally good for topics 801-850 (0.4330). Since the blocksize of our inverted block index was large (131,072), the first block of every list was always scanned and ignoring the tail of long lists did not affect the retrieval quality much, instead we gained a factor of more than 2.5 in running time. The details of the runs are given[1] in Table 2.

run	#cpu	avg query time	P@20 topics 751-800	P@20 topics 801-850
mpiiotopk	1	0.152	0.5110	0.4280
mpiiotopkpar	2	0.074	0.5110	0.4280
mpiiotopk2	1	0.057	0.4820	0.4330
mpiiotopk2p	2	0.028	0.4820	0.4330

Table 2: Performances of runs in the efficiency task: the number of CPUs used, average running time per query and precision at top-20 for our runs. The median of the average running time taken over all 25 submitted runs by all groups turns out to be exactly the same as our slowest run (0.152 sec).

9.2 Adhoc Task

Our adhoc runs were based on simple methods for constructing queries from the topics provided. For this task also we used the BM25 scoring model default parameters. For all runs, queries were processed automatically using the exact top-k algorithm same as our efficiency runs (e.g. as in **mpiiotopk**). For the four runs, the queries were constructed as follows:

- **mpiirtitle** - words in the title fields were taken as keywords.

- **mpiirdesc** - words in the description fields were taken as keywords.

- **mpiircomb** - words in the title as well as in the description fields (with possible repetition) were taken as keywords.

- **mpiirmanual** - only the construction of the queries were manual, as the keywords were chosen manually by only looking at the title, description and narrative fields.

Among these runs, the precisions of **mpiircomb** and **mpiirmanual** are better than the other two runs, as we see in Table 3.

10. CONCLUSIONS

Our focus this year was on the efficiency track. Telling from the statistics posted by the TREC organizers, our runs performed very well. Our slowest (single-processor) run was the median of all runs and our fastest run (processing 4 streams, but with only two processors, average query time: 0.028 sec) was close to the best run (average query time: 0.0125 sec). In all our experiments, most of the data was read from disk (as opposed to from main memory). The

[1]Note that although our machine had two CPUs, for the runs mpiiotopk and mpiiotopk2, a single program processed 100,000 queries sequentially, so we write that only one CPU was used.

run	P@20	bpref	map	infAP
mpiirtitle	0.4270	0.2849	0.1805	0.1678
mpiirdesc	0.4240	0.2968	0.1743	0.1471
mpiircomb	0.5020	0.3146	0.2174	0.1876
mpiirmanual	0.4810	0.3041	0.1981	0.1692

Table 3: Performances of runs in the adhoc task by P@20, bpref, map and infAP measures, averaged over topics 801-850.

precisions of our runs were decent, but could be improved by more advanced scoring models, without compromising efficiency.

11. REFERENCES

[1] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum. Io-top-k: Index-access optimized top-k query processing. In *VLDB*, pages 475–486, 2006.

[2] K. C.-C. Chang and S.-W. Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD 2002*, pages 346–357, 2002.

[3] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.

[4] U. Güntzer, W.-T. Balke, and W. Kießling. Towards efficient multi-feature queries in heterogeneous environments. In *ITCC 2001*, pages 622–628, 2001.

[5] N. Lester et al. Space-limited ranked query evaluation using adaptive pruning. In *WISE 2005*, pages 470–477, 2005.

[6] A. Marian, N. Bruno, and L. Gravano. Evaluating top-k queries over web-accessible databases. *ACM Trans. Database Syst.*, 29(2):319–362, 2004.

[7] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.*, 14(4):349–379, 1996.

[8] S. Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *ICDE 1999*, pages 22–29, 1999.

[9] S. E. Robertson and S. Walker. Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In W. B. Croft and C. J. van Rijsbergen, editors, *SIGIR*, pages 232–241. ACM/Springer, 1994.

[10] M. Theobald, G. Weikum, and R. Schenkel. Top-k query evaluation with probabilistic guarantees. In *VLDB 2004*, pages 648–659, 2004.

[11] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 2nd edition, 1999.