Looking at Limits and Tradeoffs: Sabir Research at TREC 2005

Chris Buckley
Sabir Research, Inc.
cabuckley@sabir.com

1 Introduction

Sabir Research participated in TREC-2005 in the Terabyte, Robust, and document retrieval part of the Question Answering tracks. This writeup focuses on the Robust track, and in particular on a "routing" run that took advantage of relevance judgements for the topics on the old trec V45 collection to construct queries for the new Aquaint collection. The smart_retro tool is described which given a query and the set of relevant documents, constructs an optimally weighted version of the query. smart_retro is also used to examine the differences in difficulty between the V45 and Aquaint collections (the Aquaint collection is considerably easier).

The final part of the paper describes the compression algorithms and tradeoffs that were used in both TREC 2004 and 2005. These were presented in the TREC 2004 speaker session, but never formally written up.

The hardware used for all runs was a single commodity PC with a total cost of \$1600: \$540 for a Dell PC, \$520 for four 250 GByte disks, and \$500 to bring the memory up to 2.5 GBytes.

The information retrieval software used was the research version of SMART 15.0. SMART was originally developed in the early 1960's by Gerard Salton and since then has continued to be a leading research information retrieval tool. It continues to use a statistical vector space model, with stemming, stop words, weighting, inner product similarity function, and ranked retrieval.

2 Robust

The 2005 Robust track topics consisted of 50 topics which had previously been run on the V45 documents (with Congressional Record documents removed). These 50 topics had been categorized as difficult topics on V45 by NIST.

I ran standard SMART retrieval runs (ltu query weights, Lnu document weights) including straight retrieval with no expansion, and blind feedback runs where terms occurring in

Table 1: Official Robust MAP Results

Runname	Query terms	MAP
sab05rot1	title	0.180
sab05rot2	title and blind fdbk exp	0.229
sab05rod1	description	0.184
sab05rod2	desc and blind fdbk exp	0.229
sab05roa1	all topic	0.230
sab05roa2	all and blind fdbk exp	0.255
sab05ror1	250 terms from V45 rel docs	0.262

the top documents of an initial retrieval are assumed to be useful and automatically added to the query. These approaches have been described multiple times in the past and are not further discussed here.

Overall results are given in Table 1 and were mediocre. Runs using only title words and runs using only description words performed about the same; using the entire topic did better. Blind feedback helped significantly for all three runs. Analysis revealed a bug in the blind feedback procedure that had a small effect (a mis-set parameter inadvertently combined the similarity of the initial retrieval and final retrieval), and it is also clear that I need to revisit document length normalization: there were definite biases in favor of retrieving long documents. Correcting these biases here gives roughly a 10% improvement, but further work is needed. In any case, blind feedback expansion by itself is no longer competitive with other expansion techniques (eg web expansion); the runs were made because they were simple to do and provide a reasonable base case to start future work on.

Part of the robust track effort is to predict the difficulty of the topics: rank the 50 topics in order of expected MAP score. The approach taken in both TREC 2004 and 2005 is the use of the measure topdoc_map (formerly known as anchormap) as described in SIGIR 2004 [1]. In this application, topdoc_map takes the top 30 documents of the initial retrieval run, and measures whether they stay near the top of the retrieved list after blind feedback is done. The algorithm used is that of the normal MAP measure, but with the top 30 documents of the initial retrieval considered the relevant documents for MAP. Note that no actual relevance information is used at all.

The idea is that if the blind feedback expansion is stable and leaves the initial documents near the top of the final ranked list, then both the initial retrieval and the expanded retrieval have been able to consistently capture some conceptual idea, hopefully that of the topic, and performance should be good. On the other hand, if the expanded retrieval documents have diverged significantly from the initial documents, then even the initial retrieval probably didn't capture the notion of the topic, much less the final retrieval being reasonable.

Again, results of topdoc_map for prediction are mediocre. At this time no further analysis of it has been done, but I feel there is still strong potential for prediction here (perhaps in combination with con_map described later).

2.1 Robust Routing

The Robust track this year offered a unique opportunity to look at the effect of running the old queries/topics on a new document collection that was not the intended target of the original topics. The old V45 and new Aquaint documents are similar in many important ways, such as being primarily newswire, but differ in others such as the time-span covered, and focus of the newswire sources (the V45 sources all had different emphases, while the Aquaint sources were chosen to have overlapping emphases).

This setup allows the possibility to have what was called a "human information" run: use the known relevant documents on V45 to fashion a query to run on Aquaint. This is very similar to "routing" runs in early TRECs, except there was typically more continuity / similarity of collection back then. The question is then how to fashion the query.

2.2 Robust Routing smart_retro

I've been playing around with my smart_retro tool for several years now. smart_retro attempts to find the optimal weights for a given query retrospectively, given the relevant documents for the query, a document indexing for the collection, and use of the inner product similarity function. It basically is a highly beefed up DFO (Dynamic Feedback Optimization) algorithm as described in SIGIR 1995 [2]. It's a hill climbing approach that tries increasing and decreasing query weights in turn and seeing whether MAP increases. It works quite well - given enough terms from relevant documents it can achieve a MAP value of .98 (on the 125 odd topics of V45). It also seems to be very robust in practice: on much shorter queries such as title or description queries, changing the starting conditions 25 times by randomly permuting either order or initial weights on the input query improves MAP by only 1% over all the topics. It was used as an investigative tool in the RIA workshop [4].

For this particular application, for each query I took the the 250 terms that occurred most frequently in relevant documents for that query in V45. This generally included almost every "near-stopword" in the collection (which is why as many as 250 terms were used), but in general did not include those terms which occurred in only 1 relevant document (except for a couple of topics with very few relevant documents). These terms were then weighted equally (this is discussed later), and fed into smart_retro. After about 3 minutes per topic, smart_retro would output optimally weighted (for V45) queries. These queries were then run on a version of Aquaint indexed with the same dictionary as was used for V45. The results were submitted as the sab05ror1 official run.

2.3 Robust Routing Results and Analysis

Table 1 shows that the optimized query gives a MAP score of 0.262 when run on Aquaint, just a bit better than using blind feedback with no relevance information (0.255). What's happening?

Looking at the topic by topic results, the routing run gives the best submitted result of any run on 9 out of the 50 topics. However, it also does miserably on a large number of very easy topics. For example, on Topic 325 it gets a MAP of .054, while the best run of anybody is .723, and simply running the full topic with no expansion or relevance information gives .543 (sab05roa1 run). Similarly, on Topic 433, my routing run has a MAP of .0002, the best run is .798, and sab05roa1 gets .372.

The problem is obviously overfitting, tuning the query too specifically to the relevant documents of V45. But that was expected to be the problem going in, and steps were taken to minimize the effect of overtuning (such as considering the most frequently occurring terms in the relevant documents, and not all the terms). However, looking at the queries, it becomes clear that a major problem was not over-fitting as much as a peculiar under-fitting, or under-specification.

Running the 50 optimized queries on V45 evaluate to an extremely high MAP score of 0.8995. Obviously a large number of queries had perfect, or near perfect retrieval. That is good, except it ignores how easily that perfection was achieved. In Topic 433 (mentioned earlier), for example, the optimization algorithm started with 250 equally weighted terms, and in its first pass dropped 70 terms to weight 0.0. That happened to be sufficient to give perfect retrieval - the 13 relevant documents were all retrieved first, so the algorithm couldn't improve any more and stopped. The resulting query consisted of 180 equally weighted terms, most of which were quite high frequency (about half occurring in at least 25,000 documents, and only 10 occurring in less than 240 documents). The topic was too easy.

The same effect happened on a good number of topics, though not quite as blatently: there were very large numbers of equally weighted terms indicating that an optimum query was decided without the need to distinguish between those terms.

One quick fix for this problem is to try and start with reasonable query weights before the optimization. Giving the input query terms standard Rocchio weights indeed helps a bit, giving .2878. But it doesn't solve the problem. For example, performance on Topic 433 is still only 0.0019. Some other approach is needed to avoid having the problem of too much good information to distinguish between. My current thought is simply to start with the original terms in the topic, and, one at a time, add the most frequently occurring terms in the relevant documents until the optimization algorithm can get some target score like .90 MAP.

Overall, the question of how to construct a good query that doesn't overfit the relevant documents is an important one, occurring in tasks such as filtering, routing, relevance feedback, and blind feedback. The DFO/smart_retro optimization here works very well for some topics, but fails on other easy topics and needs some more work.

2.4 Robust Collection Comparison

One interesting question in the Robust track is "What are the differences in retrieval performance caused by the change in document collection?" smart_retro can be used to measure the difficulty of the topics in a reasonably system independent way. smart_retro will give results independent of query weighting approaches (the big difference between bag-of-words systems), but document weighting (especially length normalization) and text tokenization/stemming will have a slight effect, and an inner product similarity function is

Table 2: Robust V45 vs Aquaint MAP Comparisons

Coll	title		Description	
	ltu.Lnt	smart_retro	ltu.Lnt	smart_retro
V45	0.1030	0.1421	0.1037	0.2314
Aquaint	0.1976	0.2344	0.1797	0.3310

assumed.

Table 2 gives the MAP results on the two collections of running both smart_retro, and a standard SMART inner product run (using a different document length normalization method than was used for the official runs). All runs use only either title words or description words, with no expansion.

The Aquaint collection is clearly "easier" than V45 in some sense. Well-weighted title (or description) words do a much better job at retrieving relevant documents on Aquaint. There are several possible explanations:

1. Aquaint is a more standardized newswire then V45. FBIS in particular in V45 is stranger. Vocabulary usage could be more standardized in Aquaint simply because of passage of time.

2. (Related to above) V45 might have a wider range of articles than Aquaint. Thus title words might be used in more non-relevant contexts in V45.

3. MAP has a tendency to give higher scores if it has more relevant documents, and Aquaint has more relevant documents. But our 2004 Bpref paper [3] suggests this effect is much smaller than the 40-70% increase here.

4. Aquaint is larger than V45 (twice as many documents), and could easily effectively be considered much larger. The varied subject areas of the V45 subcollections meant that many topics were restricted to one subcollection for relevant documents. The larger collection with judgement pool size remaining the same might mean that the obviously relevant documents may be dominating the judgement pool.

5. Our retrieval systems could be becoming more standardized as they improve. We could all be finding the same easier relevant documents (those that contain the title words), and the harder relevant documents may not be entering the judgement pool at all.

The last two possibilities are worrisome. It is the case that the judgement pool included a couple of runs like my routing run which used human information, and the HARD track runs, with a limited user interaction, also contributed to the pool. So it is not just BM25 and related automatic runs that supplied documents to the pool. However, I think we need to watch out for these possibilities in other TREC tracks such as the Terabyte track.

Table 3: Official Terabyte MAP Results

Runname	Query terms	MAP	Bpref	total time (sec)
sab05tbt	title only	0.1670	0.2437	28.0
sab05tball	all topic	0.2087	0.2786	135.6
sab05tbas	all, stop early	0.2088	0.2791	75.7
sabtb05ef1	title(all)	0.0411	0.0410	10198.9

Further post-TREC investigation shows that out of the 2750 documents the sab05ror1 run that were judged in the final judging pool (the top 55 documents for 50 topics), 405 unique relevant documents were found; i.e., the run was the only run to find those relevant documents in the top 55 documents. That's an unprecedently large number of unique relevant documents for a run that did not have a human oracle in the loop judging document relevance. Thus a routing run that described the learning set relevant documents instead of the topic, found a very different subset of the relevant documents than the other topic-based runs. That would seem to indicate that a substantial bias exists in the judgement pool.

3 Question Answering Track

I submitted one very basic run to the document retrieval subtask of the QA track. This was a standard SMART ltu.Lnu inner prodcut run with no expansion. The only thing unusual with the processing was that queries were first translated to standard TREC ad hoc query format, and all previous questions in the dialog were included. The run evaluated to a MAP score of 0.3197, but I've done no analysis of the results.

4 Terabyte Track

I participated in the Terabyte track with pretty much the same system and indexing as was used in the TREC 2004 Terabyte track. Since I didn't describe the system at that time, I will discuss the compression algorithms and hybrid dictionary approach used after I discuss the results.

Table 3 gives the MAP and Bpref scores, plus the total retrieval time for all topics, for the 3 ad hoc runs plus the single efficiency run. Note that the efficiency run is evaluated over 20 documents instead of 1000, and the retrieval time is over 50,000 topics instead of 50.

All runs were standard SMART ltu.Lnu runs with no query expansion, run on the nibble compressed inverted file (nibble compression described below). The only semi-interesting run from a retrieval standpoint is the "stop early" run, which ran on the full terabyte topics, but stopped retrieval as soon as the top document was guaranteed to be retrieved within the top 10 documents. This run was almost twice as fast as if the retrieval had gone to completion, but at no cost to retrieval effectiveness (even a very slight insignificant gain).

4.1 Terabyte: Hybrid Dictionary

The major change in basic indexing for the Terabyte Track was in the dictionary, mapping strings to internal concept numbers. Given the size of the collection and the uncertain nature of the documents involved (for example, many documents of binary data instead of text), it was too difficult to treat all terms as first class objects in SMART. Instead, a hybrid dictionary approach was used. There was an 800,000 entry fixed dictionary of words and stems; if a token in a document occurred in the dictionary then the corresponding concept number (between 1 and 1,000,000) of the token's stem was assigned. If the token did not occur in the fixed dictionary, then a hash value for the token was computed, between 1,000,000 and 3,000,000, and assigned.

The major disadvantage of the hybrid dictionary is that multiple different tokens that did not occur in the fixed dictionary could be assigned the same concept number. Thus a user query using one of these tokens might retrieve additional unwanted documents due to these false matches. In practice the chance that a random document containing a false match would also match the rest of the user's query is very small. In addition, the more advanced search modules of SMART re-index the top documents, and can detect the false match.

The advantages of the hybrid dictionary approach are that it is fast, and every token can still get indexed. Thus there is no reason to filter out foreign language or binary data. Indeed, if a user has a binary data chunk from some source, they can use that chunk as a query and find other occurrences of that binary chunk in the collection.

4.2 Terabyte: Compression

The major effort in indexing the Terabyte collection was in compressing inverted lists in the collection index. The goal of index compression is to reduce the amount of space needed to store inverted lists, at a cost in indexing and possibly a cost in retrieval time.

A typical inverted list gives the documents which contain a given term along with the weight of the term in that document. Ignoring the weight for now, the list for a common term might look like $(2, 5, 7, 13, ..., 2300001, 2300004, 2300008, 2300009)$

The numbers in that list can be represented in skip-list form as the differences between successive entries, rather than the entries: $(2, 3, 2, 6, ..., 3, 4, 1)$

Just as it takes less room on the page to write out the list, it can take many fewer bytes to store a list of 10,000,000 small numbers rather than 10,000,000 large numbers. There have been numerous papers on methods to encode these long lists of small numbers; the reader is referred to Managing Gigabytes by Moffat et al[6] for a good overview.

Silvestri et al in SIGIR 2004[5] had a very nice approach where they clustered the collection and then renumbered it. Documents which shared lots of common terms were therefore given very close document numbers and therefore could be represented as very small differences in the term skip-lists.

The major fault of that approach is that it treats all terms and documents equally. It's not greedy enough. Zipf's law states that most of the occurrences in the inverted lists will come from the very common terms. Renumbering is a good idea, but a more greedy approach would be to guarantee the longest lists will be compressed the most.

4.2.1 Adjacent Docid Renumbering

The goal of Adjacent Docid Renumbering is to maximize the number of adjacent document numbers (a difference of 1) in the inverted lists. If skip-lists have long sequences of 1's, then they should be easy to compress.

The approach I used is to be as greedy as possible. Start with the longest inverted list, make sure as many entries in that skip-list are 1 as possible, then consider the next longest inverted list, and continue until all inverted lists are done.

Represent the longest inverted list as a bit-vector in the total list of documents

0100101000001010100111001000...0110100010101001100100011

Renumber (move) all the documents represented with a 1 to the beginning of the list, forming two buckets. The first bucket will contain all documents with the longest list term present, and the second will contain those documents without that term. A skip list made now for the longest list term will contain all 1's.

111111111..1111111111111_000000000..0000000000000

Take the next longest inverted list. Do the same thing within each of the two buckets, forming four buckets.

1111..111111_0000..00000_00000000..000_11111..111

Since documents are only moved within the already formed buckets, a document movement does not alter the number of adjacent docids in any preceding inverted list. The longest list term will always have a skip-list of all 1's.

Note that when a bucket is split into two parts, alternating between putting the 1's first or the 0's first may mean longer sequences of 0's or 1's.

Continue splitting buckets until all inverted lists of length greater than 2 have been considered. The final renumbering has now been done. and each docid can be mapped to their final values.

The actual implementation within SMART does not actually change the docid assigned to any document. Instead the mapping of docids is used to create the compressed inverted index, and after inverted file retrieval the compressed docids are re-mapped out to their original values. This allows an overall simpler system design for dynamic collections.

The implementation of Adjacent Docid Renumbering is a bit more complicated than the conceptual algorithm above. If after the first 20 inverted lists, there are 2^{20} buckets formed, it is quite inefficient to go through the entire 21^{st} inverted list once for each bucket. Thus the actual algorithm iterates over the inverted list, and splits the bucket that each inverted list entry occurs in.

The final implementation was reasonably efficient. It took 6 clock hours to construct the docid map (and its inverse). The running time in practice is dominated by the time needed to sort each inverted list.

$O(NlogN + KlogK + \sum_{k=1-K}(\ |k|log\ |k| \ ))$

where K is the number of inverted lists, N is the number of terms, and $|k|$ is the number of terms in inverted list k.

This greedy renumbering scheme worked amazingly well. There were 4.3 billion postings in the inverted index for the Terabyte Collection. After Adjacent Docid Renumbering, 2.96 billion postings were adjacent (skip-list value of 1).

I tried several attempts at further optimizing the number of 1's. One to two percent can be gained in compression by instead of using the next longest list as the next bucket splitter, use the list with the most docids in common with the current list. The gain was more substantial on smaller versions of the collection, but was more marginal on the full collection, and almost doubled the running time.

4.2.2 Compressing After Adjacent Document Renumbering

There are more possibilities for compression algorithms when over 2/3 of the skip-list entries are 1. I tried two major variants:

1. Multiple custom scheme: Try 15 different compression schemes (each optimized for a different type of skip-list) and choose the best on a per list basis.

2. Nibble scheme: All encodings were on a half-byte (nibble) boundary for efficiency.

The multiple custom scheme was designed for optimal compression, at a cost in indexing time and retrieval time. Each compressed inverted list contained a 4 bit header indicating which of the 15 compression methods was used for this list, followed by up to 52 bits giving parameters for that particular method. Figure 1 gives a sample complicated compression method designed for long lists with broken sequences of 1's. Using that scheme, conceivably the next 256 docid entries in an inverted list could be represented by one bit: if the fifth choice out of the eight is given the prefix of zero 0's, and the next 256 entries in the skip-list are all 1.

The nibble scheme was designed to be more efficient at the cost of less compression. Arbitrary bit-by-bit operations are expensive on PCs. It's much more efficient to get and operate on entire bytes (8 bits). The nibble is a compromise; you get a byte at a time but operate on one of the first half, or the second half, or the entire byte. Figure 2 describes the nibble scheme. All encodings start and end on a nibble boundary so it is considerably faster to both compress and uncompress.

In addition to the document numbers, compressible by one of the two above schemes, the document weights also have to be compressed in the inverted lists. Since the SMART architecture allows for reasonably arbitrary weights, the weight compression scheme has to be fairly general, and can't be optimized strongly. The approach used here is to store the minimum and maximum weights in a given inverted list as two 4-byte floating point numbers at the start of the compressed list. Then each document weight is stored in some fixed N bits per weight linearly scaled between these minimum and maximum weights. If N is 6, then there are 64 possible values for document weights in this list. If N is 4, then there are 16. If N is too low, then there is not sufficient power to distinguish between document weights

Each compressed entry consists of a prefix giving 8 choices describing the entry, and possibly a following docid:

- Next docid has a skip-list entry of 1.
- Next 4 docids have a skip-list entry of 1.
- Next 16 docids have a skip-list entry of 1.
- Next 64 docids have a skip-list entry of 1.
- Next 256 docids have a skip-list entry of 1.
- Next docid is represented in ıthresh1 bits.
- Next docid is represented in the following ıthresh2 bits.
- Next docid is represented in the following ıthresh3 bits.

The prefix is a sequence of zero to seven 0's followed by a one (except after seven 0's). The initial parameter bits consists of three 5-bit numbers giving the thresh1, thresh2 and thresh3 values, plus seven 3-bit numbers giving the prefix for each of the first 7 choices above. The threshholds and the prefixes are optimized for representing this inverted list.

Figure 1: Sample complex compression scheme (OneOctDyn)

Each compressed entry consists of a nibble giving 16 entry formats, possibly followed by a docid. Eight format choices give a number of sequential skip-list entries of 1 being represented (1,2,4,8,16,64,512,4096). Eight format choices give the number of nibbles taken by the following docid (1-8). It is assumed all renumbered docids can be represented in 32 bits. There are no parameters in the nibble scheme.

Figure 2: Nibble Scheme

Table 4: Compression Tradeoffs

	Compression		Retrieval			
	Size (*10e9)	Time (Sec)	Time(Cold) (Sec/q)	CPU Time (Sec/q)	Page Faults (4K Pages)	BPref
None	32.37	0	7.54	1.68	3,649,523	.206
Full (6bit wts)	5.07	27431	2.61	1.73	441,196	.206
Full (4bit wts)	4.00	34955	2.61	1.69	326,963	.203
Nibble	4.40	13724	1.99	1.18	357,463	.203
None(Title)	32.37	0	1.62	0.36	452,471	.151
Nibble(Title)	4.40	13724	0.32	0.27	58,255	.147

and retrieval will suffer. If the nibble scheme of compressing docids is used, the current implementation, which stores docids and their weights together for efficiency reasons, must use a value of N=4.

Table 4 gives both indexing and retrieval figures for the various compression schemes. The first four lines compare four compression schemes (No compression, Full compression with 6 bit weights, Full compression with 4 bit weights, and Nibble compression with 4 bit weights), run on long topics. The last two lines give two compression schemes on very short topics (title only).

The second column indicates the amount of compression. The full compression, unsurprisingly, does the best. On average it takes 3.4 bits per docid. One interesting fact not in the table is that the 15 compression schemes comprising the full compression only do about 5% better than the best single scheme (the OneOctDyn scheme described previously). However, as the collection changes size, the 5% difference remains about the same, but the best single scheme changes. OneOctDyn is no longer optimal on one twentieth sized subset of the Terabyte collection. So the full compression scheme is more general even though single schemes may do well enough on a particular collection.

In practice, the nibble scheme is fast and compresses enough to be useful in most general applications. It's been used in several internal investigations. Note that Table 4 shows that the CPU time using the nibble approach is considerably faster than the CPU time using no compression whatsoever. Part of that appears to be the effect of memory cache, especially keeping the partial accumulation sums of the standard inverted search algorithm in the cache. The long inverted lists with many adjacent docids in a row means the same cache line containing partial sums will get hit multiple times in a row without having to go back and forth to main memory. We do not get that savings with the no-compression search, or in the nibble search if we translate back from the document renumbering to the original docids before we do the partial sum accumulation

5 Conclusion

Sabir Research participated in a number of tracks in TREC 2005. The major points described here that might be of interest to the wider IR community are

- The Robust Track routing run sab05ror1 which worked extremely well on some topics and quite poorly on others due to at least two different sorts of over-fitting.
- sab05ror1 also found a large number of relevant documents in its top 55 ranks that no other run found. That suggests a possibility of bias in the document judgement pool due to the size of the collection.
- The Sabir Research Terabyte Track runs were fast, but only mediocre in effectiveness. The efficiency topics took about 0.2 seconds per topic.
- A hybrid dictionary was used for the Terabyte indexing in both 2004 and 2005, which indexed all terms (except stop words) in all documents, including binary documents. It was fast and worked well.
- A new compression algorithm was used in 2004 and 2005 that compresses docids better than previous algorithms. It does a very aggressive "Adjacent Document Renumbering" to get as many entries to be 1 as possible in the inverted file skip-lists, and then compresses these '1's effectively. After renumbering the Terabyte collection, over 68% of the skip list entries are '1'.
- Search on the compressed inverted file was much faster in elapsed time, and even considerably faster in pure CPU time than on the original inverted lists.

References

[1] C. Buckley. Topic prediction based on comparative retrieval rankings. In K. Jarvelin, J. Allan, P. Bruza, and M. Sanderson, editors, *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 506–507, 2004.

[2] C. Buckley and G. Salton. Optimization of relevance feedback weights. In E. Fox, P. Ingwersen, and R. Fidel, editors, *Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 351–357, 1995.

[3] C. Buckley and E. Voorhees. Retrieval evaluation with incomplete information. In K. Jarvelin, J. Allan, P. Bruza, and M. Sanderson, editors, *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 25–32, 2004.

[4] Chris Buckley and Donna Harman. Reliable information access final workshop report. ARDA Northeast Regional Research Center Technical Report, 2004.

[5] F. Silvestri, S. Orlando, and R. Perego. Assigning identifiers to documents to enhance the clustering property of fulltext indexes. In K. Jarvelin, J. Allan, P. Bruza, and M. Sanderson, editors, *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 305–312, 2004.

[6] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes - Compressing and Indexing Documents and Images, Second Edition.* Morgan Kaufmann Publishing, 1999.