IXE at the TREC 2005 TeraByte task

Giuseppe Attardi
Dipartimento di Informatica
Università di Pisa – Italy

Abstract

The TREC Terabyte task provides an opportunity to analyze scalability issues in document retrieval systems. I describe how to overcome some of these issues and in particular improvements to the IXE search engine in order to achieve higher precision while maintaining good retrieval performance. A new algorithm has been introduced to handle OR queries efficiently. A proximity factor is also computed and added to the relevance score obtained by the PL2 document weighting model: several experiments have been performed to tune its parameters. By tuning also other parameters used in relevance ranking, IXE achieved second best overall P@10 score, combined with the fastest reported retrieval speed.

1. Introduction

The purpose of the TREC Terabyte track is "to investigate whether/how the IR community can scale traditional IR test-collection-based evaluation to significantly larger document collections than those currently used in TREC". However disk capacities and processor performance have increased enough that a collection of the size of the GOV2 collection can be handled on a single PC. In fact most of the participants in TREC 2005 used systems ranging from 1 to 2 CPUs, except for Peking University (16), University of Melbourne (8), RMIT (8), Dublin City University (5).

My goal was to investigate indeed the issues of scaling to a distributed system of realistic size, so I experimented with the largest set of CPUs I could afford, irrespective of the individual performance and disk size. In fact I used machines with fairly slow CPU speed (800 MHz Pentium III), small and slow disk (60 GB at 4200 rpm), slightly compensated by 1 GB of RAM each. The software is based on the IXE search engine, which is a full blown Web search engine, designed to handle hundreds of queries per second on a single PC and providing integrated multithreaded search capabilities, a customizable query language, an extensible document reader architecture and an object store for document storage.

My experiments with such architecture at TREC 2004 showed that with a properly designed distributed system and careful tuning of several parameters, indeed the performance can scale linearly, and the system achieved the second best retrieval speed performance. However there were unexpected effects on relevance, which I have addressed this year as follows:

- a specialized algorithm has been introduced to handle OR queries, which are considered the most effective for the ad-hoc task
- a proximity factor has been added in the computation of the relevance score
- the global IDF across all the sub-collections is computed and distributed to each search node.

The effects of these changes were almost a doubling of P@10, achieving a score of 0.678, which ranks IXE close second to the best result of 0,690. IXE performance, measured as average time to retrieve to 20 results, was 0.14 sec, as compared to 29.46 sec for the other mentioned system, which employed 2 CPUs. Among all participants, only the University of Melbourne reported a performance below 1 sec (i.e. 0.2 sec) in the ad-hoc task.

In the rest of the paper I will discuss these solutions and describe the experiments performed in order to tune the parameters used in the evaluation.

2. Search System Architecture

A cluster of Linux blade servers was used to build a high performance distributed search service on the GOV2 collection. The hardware consists of an RLX 300X chassis, filled with 24 800xi blades, whose current cost is about $ 10,000. One of the blades is dedicated to the role of control tower and is used to manage the cluster. The remaining 23 blades are used as server blades for indexing and search: they run the Linux Fedora Core 1 release. The overall amount of disk available is about 1.3 TB and the amount of RAM is 23 GB.

2.1. The IXE Search Engine

For indexing and querying the collection, I used IXE (IndeXing and search Engine) a C++ library for developing search applications, which I had also used in TREC 2004. IXE provides a wide range of facilities for document handling, from tokenization to regular expressions, to multiple encoding, which can be extended through its pluggable document-reader architecture. IXE allows creating high throughput search services by means of integrated threading and HTTP support. The library has been designed for implementing high performance full-text search services, but is capable of handling more complex structures than pure text documents by means of an object store that provides persistency to C++ objects.

Query processing is handled though a ***Query Cursor Interface***, which can be extended through the object-oriented mechanisms of class specialization and polymorphism. A query gets compiled before execution into an appropriate combination of `QueryCursor`'s, each one handling a subquery. For instance queries like

```
text matches sun moon       // pages containing sun and moon
text matches proximity 20 [‘sun flower’ moon] // moon must appear within 20
words from ‘sun flower’
```

are compiled to:

```
QueryCursorAll(QueryCursorWord(“sun”), QueryCursorWord(“moon”))
QueryCursorProximity(20, QueryCursorAnd(QueryCursorPhrase(“sun”, “flower”),
QueryCursorWord(“moon”)))
```

A `QueryCursor` provides an iterator interface to retrieve results form the query on demand, rather that as a bulk operation that fills a whole table of results. The application is so given the possibility of deciding how to use the results while they are generated, without having to wait for all of them to be collected. The iterator interface is the following:

```
QueryResult* QueryCursor<Collection>::next(QueryResult& min)
```

`QueryResult` is a class describing a single result, i.e. a quadruple consisting of a collection id, document id, word position and relevance score. `QueryResults` are totally ordered in a natural

lexicographic order of its components (except the score). The `next(min)` method returns the next results which is greater or equal than `min`. This generalization allows handling different type of requests with a single interface: for instance one can decide whether to use or not position information in looking for the occurrence of a word in a document.

For cursors which combine results from other cursors by performing their intersection, IXE exploits the Small Adaptive Set Intersection algorithm [3]. This algorithm requires results to be returned in increasing order. This constraint applies to all cursors, since they can be combined arbitrarily in queries. Cursors on words satisfy this constraint since they scan posting lists which are stored sorted by ***docid***. This approach provides excellent performance and low memory use for And queries, which are most frequently used in Web search engines. In [1] I reported on benchmarks comparing IXE with Zettair and Lucene on such kind of queries.

For ***ad hoc*** retrieval though OR queries are preferable. Insisting on returning results sorted by docid has the effect that the presence of a frequent word in a query forces to sift through a large number of results. However for practical reasons it is typical to set a cutoff on the number of results considered (e.g. 20,000), but this may lead to discard highly relevant documents which just happened to have a high ***docid***. A better strategy is to perform the union of the posting lists starting from that of the term with highest IDF: documents with higher relevance score are more likely to be returned first and if the cutoff is reached only documents containing just frequent words will be discarded. I implemented this strategy as a specialization of `QueryCursorSome` when all sub-cursors are of type `QueryCursorWord`. To avoid returning duplicates, I add a bitmap to each sub-cursor, with as many entries as the length of its posting list. When a result with docid ***d*** is obtained from one sub-cursor, I mark its position bit in all the following cursors: this requires advancing those cursors up to the first result greater than ***d***. All sub-cursors will now point to document ***d***, if they contain it. Hence I can compute the relevance score for ***d*** by adding the score from all sub-cursors that do. These cursors will be reset to the beginning when the iterator will move to the next sub-cursor. When iterating through a sub-cursor, marked results are skipped.

This algorithm turned out to be twice faster than another common solution of maintaining a limited set of accumulators (e.g. 20,000) mapping result ***docid***s to their corresponding relevance score. Whenever a ***docid*** is returned from one sub-cursor, its score is added to its accumulator. Apparently the cost of accessing the accumulators through hash tables overcomes the cost of scanning cursors repeatedly, which is speed up by means of skip lists.

The relevance score of a result is computed only after the iterator for the whole query has returned: this avoids computing scores for documents which will be discarded for instance for failing a join. In the runs I used the PL2 measure [5], which performed better than BM25 and cosine measure in our tests on the 2004 qrels.

I expected to improve precision by incorporating a factor in the score to represent proximity. To compute the proximity score I applied the algorithm by Song et al. [7], which computes a series of non overlapping spans of query terms occurrences in a document. Each term is given a weight according to the size of the span in which it occurs. The relevance contribution of one term occurrence is given by:

$$f(t, span_t) = \frac{n_t^y}{|span_t|^x} \cdot \frac{pf}{|q| - n_t + 1}$$

where:

t is a query term,

$span_t$ is a maximal non overlapping span that contains ***t***,

n_t is the number of query terms that occur in $span_t$,

$|span_t|$ is the width of $span_t$,

$|q|$ is the length of the query,

x is an exponent that is used to restrain that the value decayed too rapidly with the density of a span increasing (0.09),

y is an exponent to decrease the influence of the size of a span (0.19),

pf is a factor used to dampen the overall influence of proximity in the relevance score.

The parameters ***x*** and ***y*** where tuned on the TeraByte 2004 qrels to produce the following values (n_t in rows, span width in columns):

width	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1																			
2		1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1	1	1	1	1	1	1	1	1	1	1
3			1.3	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.1	1.1	1.1
4				1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.2

Song reports improvements on P@10 which range from 4.7% to 10.2% on Trec-9 and Trec-11 collections respectively, which however correspond to an absolute increase of 1% from 26% on Trec-9 and of 2.5% from 24% on Trec-11. Our experiments on the 2004 qrels showed that adding the proximity weight had often negative effects on precision. Only by damping significantly the influence of proximity, using a value of ***pf*** = 1/8, I achieved an improvement of P@10 of approximately 1%. But since this improvement is on top of a quite higher score of over 65% this is not necessarily inconsistent with Song experiments and other experiments: i.e. one can only expect marginal improvements by incorporating proximity in the relevance score.

A second variant introduces a weight according to the "color" of term, which represents term features like capitalization or appearance in certain areas of the document (title, heading, anchor, etc).

Finally, I introduced a weight for query terms, to distinguish between terms extracted from the title topics (weight 1.0) from those extracted from the description or narrative (weight 0.3 or 0.1 depending on the length of the narrative).

3. Distributed Indexing and Search

The collection is distributed by partitioning it according to the local (document based) model [6], where each server handles an inverted index for a disjoint set of documents in the collection.

For building the indexes, I used the same indexing process [1] that I employed for TREC 2004, except that I divided the GOV2 collection into 23 shards, one for each blade and created a single index for each shard, containing around 1 million documents each, taking 4 GB of disk space. Overall the index size is 92 GB, made up of three parts as listed in Table 1.

Indexing the whole collection was done in parallel on all 23 blades and took ~12 hours, 2/3 of which was due to uncompressing the collection, which was done on the fly to overcome disk space limitations. Furthermore I employed an HTML document reader that performs full HTML structure analysis, in order to assign colors to terms. Skipping this extra analysis further increases indexing performance to 24 GB per hour on a single 800MHz PC.

Data Structure	Size
Lexicon	4.2 GB
Posting Lists (including positions)	62.0 GB
Metadata	26.0 GB
Document cache (optional)	84.0 GB

Table 1. Size of the index files for the GOV2 collection.

The index is enriched with additional text for each document which consists in:

1. Anchor text: extracted from the anchors of document's in-links
2. Description text: extracted from the description of the document, either from Dmoz or from the META description tag
3. URL tokens: extracted from the document's URL.

The index stores also a type ('color') for each term. Colors are used for weighting the term in relevance ranking and include:

a) Title – for words in the document title
b) Heading – for words in the document's headers
c) Anchor – for words in the document's anchor elements
d) Description – for words in Dmoz descriptions
e) Url – for words in the document's url
f) Context – for words in links to the document.

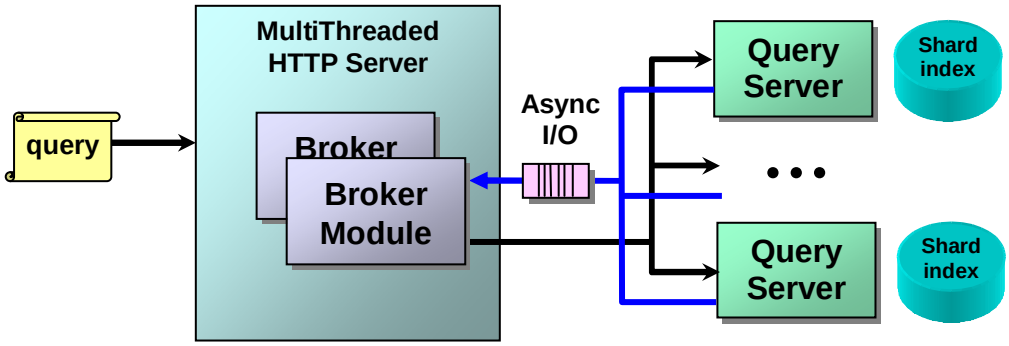
3.1. Search

Each blade runs a search service on its shard of the collection. The service is accessible remotely in two forms:

- as an XML Web Service,
- through an internal binary communication protocol.

In both cases queries are submitted using the syntax for HTTP query strings, i.e. as a '?" followed by a series of ***parameter***=***value*** pairs each of which is URL-encoded.

A query broker runs on one of the blades: its task is to accept queries, dispatch them to the query servers, collect all results and merge them into a single result list. The broker communicates with the query servers by asynchronous IO, to reduce wait and latency. The broker can be invoked in batch mode to produce the TREC runs. A broker module instead gets instantiated within a thread pool through a Web interface to provide a query interface typical of Web search engines.



Each query server is responsible for a shard, a disjoint subset of documents in the collection (called local inverted files in [6]). The local inverted file organization uses system resources effectively, provides good query throughput and is more resilient to failures.

The index for each shard is generated locally and hence it does not contain global statistics like average document length and IDF. The computation of the PL2 score involves such parameters and unfortunately the variation of these parameters in different shards greatly affects precision. Using IDF from the local indexes I could achieve a P@10 score of 49% at best. I introduced a change in the query broker to collect document frequency from all shards, compute the global document frequencies and distribute them to all indexes. Using these global statistics, P@10 grew up to 68%.

The choice of distributing the global statistics is controversial: other participants avoided doing so [personal communication], but apparently their score was not affected significantly, possibly because they were using a small number of CPUs. The significant lower relevance I measured when not using global statistics might be due to the non randomness in the composition of the shards. The shards were made dividing the collection according to the grouping in sub-collections as provided by TREC. Since the documents were grouped according to crawling order, i.e. through a breadth-first visit of the `.gov` domain, it is likely that a non uniform bias is present in the composition of documents in the various shards.

Google employs a distributed cluster architecture [4], where each document server handles a shard of randomly selected documents. Such random distribution from a huge number of documents might produce a sufficiently similar IDF values in each shard to allow avoiding a global IDF. Since reportedly Google ranking relies on over a hundred other parameters, not using a global IDF may not have such an impact on relevance as in the TREC experiments.

Relying on the use of global statistics may complicate computing the relevance score for certain special query functions, provided by a Web search engine, e.g. searches restricted to a specific domain. To properly compute relevance, TF/IDF values for such searches would require the use of global values for the implicit collection defined by the domain. Computing and storing such statistics for every possible domain would be impractical. Alternatively one would have to compute such values on the fly for each query, performing a four step process: 1) collecting the IDF for the words in the query, 2) adding them up, 3) passing these values to each query server so that it could independently compute the ranking for the results from its shard and 4) collecting

and merging the results from each shard. A similar solution is used in Terrier [5], while in Sidra [2] ranking is performed by the query broker, by collecting ranking information from the servers.

4. Results

Search was distributed on the 23 blades, collecting 1000 results from each shard, which were merge sorted retaining only the top 1000.

4.1. Ad hoc

I submitted four runs to the ad hoc task. The baseline run (pisaTTitProx) used OR queries made with just the title words, color weighting was disabled while proximity weight was included. No link analysis was used.

A second run (pisaLonProx) used instead long queries, built by OR-ing title words with a weight of 1.0 and description and narrative words with a weight of 0.3 (lowered to 0.1 for long narratives).

The third run (pisaLonProxA) was similar to the second, with the inclusion of color weightings in the relevance score.

The fourth run (pisaLonProxC) was similar to the third, with the inclusion of a page link score computed according to the number $\boldsymbol{n}$ of links pointing to $\boldsymbol{p}$, by the formula:

$$\boldsymbol{Lr}(\boldsymbol{p}) = \begin{cases} 1.0 & \boldsymbol{n} \geq \boldsymbol{N} \\ \sqrt{\boldsymbol{n}/\boldsymbol{N}} & \boldsymbol{otherwise} \end{cases}$$

where $\boldsymbol{N}$ is an upper bound on a page's in-link number.

Run	MAP	P@5	P@10	R-precision	Relevant
pisaTTitProx	0.26	0.57	0.56	0.33	6293
pisaTLonProx	0.11	0.57	0.54	0.19	3031
pisaTLonPrxA	0.29	0.68	0.66	0.37	5638
pisaTLonPrxC	0.28	0.70	0.68	0.35	5347

4.2. Efficiency Task

I submitted three runs to the TREC 2005 TeraByte efficiency task.

The first run (pisaEff2) uses OR queries, proximity weights and a cutoff of 10,000 on each shard. The average time to return top 20 documents was 0.25 seconds.

The second run (pisaEff3) is similar except that it does not use proximity weights. The average time to return top 20 documents was 0.2 seconds.

The third run (pisaEff4) uses instead AND queries but increases the cutoff to 20k. AND queries are handled more efficiently, with an average time to return top 20 documents of 0.14 seconds.

Run	MAP	P@5	P@10	R-precision	Relevant
pisaEff2	0.05	0.44	0.43	0.06	435
pisaEff3	0.05	0.47	0.46	0.06	431
pisaEff4	0.03	0.36	0.35	0.05	342

4.3. Named page

For the named page task I used AND queries from the title topics. The three submitted runs differ from the baseline (pisaTNp) with the addition of proximity weights (pisaTNpProx) and page popularity (pisaTNpProxC).

Run	ARR	Top10	Top10%	Not found	Not Fount %
pisaTNp	0.23	82	32	146	56
pisaTNpProx	0.26	90	36	138	55
pisaTNpProxC	0.27	91	36	137	54

5. Conclusions

A significant improvement in precision has been achieved with the IXE search engine in the TeraByte ad-hoc task by introducing a new algorithm for handling OR queries and adopting the PL2 relevance score formula, combined with a proximity weight, a color weight and a popularity weight. Adding a proximity factor in the relevance score however produced only a modest, less than expected, improvement.

Experiments with a medium-scale distributed architecture showed as expected linear scalability in performance (either in the size of the collection or in queries per second). However precision was affected significantly when using a purely distributed search, without use of global statistics. Computing global IDF values and distributing them to each server was necessary to maintain an overall high score in precision.

6. Acknowledgements

This research was supported in part by the Italian MIUR ministry as part of project Grid.it. Maria Simi provided comments on a draft of the paper.

7. References

[1] G. Attardi, A. Esuli, C. Patel. Using Clustering and Blade Clusters in the TeraByte task, ***Text REtrieval Conference (TREC) 2004 Proceedings***, 2005.

[2] M. Costa, SIDRA: a Flexible Web Search System. FCUL Technical Report DI/FCUL TR 4-17, November 2004.

[3] Erik D. Demaine, A. Lopez-Ortiz and J. Ian Munro. Experiments on Adaptive Set Intersections for Text Retrieval Systems, in ***Proceedings of the 3rd Workshop on Algorithm Engineering and Experiments***, ***LNCS***, Washington, DC, January 5-6, 2001.

[4] U. Hoelzle, J. Dean, and L.A. Barroso. Web Search for A Planet: The Architecture of the Google Cluster. ***IEEE Micro Magazine***, April 2003, 22-28.

[5] I. Ounis, et al. Terrier Information Retrieval Platform. In ***Proceedings of the 27th European Conference on Information Retrieva***l (ECIR 2005), LNCS, Santiago de Compostela, Spain, 21-23 March 2005, Springer.

[6] B. Ribeiro-Neto, R.A. Barbosa. Query performance for tightly coupled distributed digital libraries. In ***Proceedings of the 3rd ACM Conference on Digital Libraries***, 182-190, 1998.
[7] S.E. Robertson, K. Sparck Jones. Relevance Weighting of Search Terms, ***Journal of the American Society for Information Science***, 27(3):129-146, 1976.
[8] G. Salton and C. Buckley. Improving retrieval performance by relevance feedback. ***Journal of the American Society for Information Science***, 41(4):288-297, 1990.
[9] R. Song, J. Wen, W. Ma, Viewing Term Proximity from a Different Perspective, Microsoft Research Asia, TR-2005-69, 2005.
[10] I. Witten, A. Moffat, T. Bell. ***Managing Gigabytes: Compressing and Indexing Documents and Images***, 2nd edition, Morgan Kaufmann Publishers, 1999.