Using Clustering and Blade Clusters in the TeraByte task

Giuseppe Attardi, Andrea Esuli, Chirag Patel
Dipartimento di Informatica
Università di Pisa – Italy

Abstract

Web search engines exploit conjunctive queries and special ranking criteria which differ from the disjunctive queries typically used for ad-hoc retrieval. We wanted to asses the effectiveness of those techniques in the TeraByte task, in particular scoring criteria like: link popularity, proximity boosting, home page score, descriptions and anchor text. Since conjunctive queries sometimes produce low recall, we tested a new approach to query expansion, which extracts additional query terms from a clustering of the snippets from the first query. The technique proved effective, almost doubling the Mean Average Precision. However, the improvement was just enough to compensate for the drop that was introduced, contrary to our expectations, by the proximity boost.

1. Search System Architecture

For handling queries in the TeraByte track we built a high performance distributed search service on the GOV2 collection, based on a cluster of Linux blade servers.

The hardware consists of an RLX 300X chassis, filled with 24 800xi blades, whose total cost is about $ 25,000.

Each blade consists of an 800 MHz Pentium III processor, 1 GB of RAM, one 60 GB 2.5" hard disk (at only 4200 rpm) and three 100 Mbps network adapters. 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.

1.1. The IXE Search Engine

IXE (IndeXing and search Engine) is a C++ library for developing search applications. 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 supporting persistent C++ objects.

Support for persistency is provided through a mechanism of reflection, implemented using C++ meta-template programming [1]. Reflection creates a metaclass for each class of objects, where properties of object fields can be stored, such as the maximum size of a field value or indexing properties.

For example, a class for describing a Web page may look like this:

```
class Page {
  public:
    unsigned id;      // unique id
    char* url;        // page URL
    char* text;       // page content
    unsigned links;   // number of links in the page

  META(Page,
     (KEY(id, Field::autoincrement),   // automatically assigned object id
      VARKEY(url, Field::unique, 1023),     // indexed, value is unique
      VARKEY(text, Filed::fulltext, 65535), // indexed full-text
      FIELD(links)));                       // persistent, but not indexed
};
```

The `META` construct is used to annotate the class definition with attributes, which are exploited in serialization and indexing: for instance whether a field has unique values in a collection or whether a full-text index must be created for the field.

Objects are stored in persistent object relational tables. Objects can be retrieved from tables by means of cursors: access operations on cursors return real objects, not `ResultSets` or similar data structures (as in ADO, or JDBC), from which the program must extract values for each field with explicit access operations.

A *sequential cursor* can be created on any indexed field. *Query cursors* instead can be used to select objects which match conditions expressed in a SQL-like query language. A query expression is a logic combination of conditions on object attributes. Queries are compiled before execution and results are obtained scanning the query cursor on demand, rather that as a bulk operation that fills a whole table of results. The application is so given the possibility of performing decisions on how to use the results, without having to wait for all of them to be collected. For instance, a heap structure can be used to sort the results as they are generated, keeping just the top k [1], rather than having to sort them all: this reduces the complexity to $n \log(k)$, rather than $n \log(n)$ and reduces the amount of space required to $O(k)$ which can results in significant savings when $n >> k$.

Matching conditions on full-text attributes varies from matching simple list of words, to phrase matching, to proximity matching, and to any combination thereof.

Here is a sample of queries on a collection of `Page` objects:

```
text matches sun moon       // pages containing sun and moon
text matches 'sun flower'   // pages with the phrase 'sun flower'
text matches proximity 20 ['sun flower' moon] // moon must appear within 20
words from 'sun flower'
links > 8 and text matches 'moon light' // combination of conditions
```

Results from queries on full-text indices are assigned a relevance score based on a classical cosine measure, augmented by a factor for proximity and by 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). Computation of the relevance score can be customized for each application to take into account additional factors, as described later for the GOV2 collection.

1.2. Distribution of the GOV2 collection

The GOV2 collection consists of over 80 GB of compressed documents (426 GB uncompressed) split into 273 directories, each one containing 350 MB of documents, stored in one hundred files in *gzip* format.

We divided the collection into 23 shards, one for each blade. Each shard consists of 12 sub-collections, corresponding to directories in the distribution, for a total of around 1 million documents each, taking 4 GB of disk space.

The documents are kept compressed on disk: during indexing one file at a time is uncompressed and its content piped to the indexing program.

For convenience, we also shared data among the servers using NFS: each blade exports its `/export` directory to the other blades and mounts in the `/import` directory the directories exported by the other blades, obtaining this view:

`/export`	data exported from this blade
`/import/b1`	access to blade 1 /export directory
`/import/b2`	access to blade 2 /export directory
…	
`/import/b24`	access to blade 24 /export directory

1.3. Preparation of support data

For properly cross referencing documents, in particular for link analysis, we needed to assign a global ID to each unique URL in the collection. To accomplish this efficiently we partitioned the URLs according to a hash code and on each blade we created a table for its partition, eliminating duplicates present in the original collection. The various tables are accessed as a single table, by using the same hash code to select which partition to search. Building the URL tables in this distributed fashion took about one hour.

A table of citations (the text inside an anchor tag) was used during indexing to enrich the document's content with the often meaningful text extracted from other documents that point to it. For example, a document containing the text of a law about "Federal welfare reform" (topic 720) may not contain these terms, while other documents may mention it using those words.

Citations were extracted analyzing all documents in the collection. A citation consists of: the referrer URL id; the target URL id; and the citation text. A compact encoding was used for being able to handle large document collections. We created on each blade a table of citations to documents in other shards, making up in total a 23×23 matrix of citation tables.

The citation tables for each shard were then merged into a single table, containing all citations to documents in that shard. While a single centralized citations table would have had a size of around 28 GB, with this approach we created instead a citation table of 1.2 GB on each blade with only the required data for indexing the shard on that particular blade.

Creating citations tables in a distributed way allowed us to reduce the processing time to less than 5 hours.

Additional descriptive text to be added to each document contents was obtained from the Dmoz archive.

The Dmoz [6] archive is a hand made Web directory, that contains over 3 million URLs classified within several hundred categories. We extracted the description associated to each URL and its classification categories and stored them in an IXE table. During indexing, the text of the description extracted from this table is added in a field of the `PageInfo` object that represents the URL. The text in this description gets indexed as part of the document, but it is given a special color that produces a higher weight in the retrieval score.

1.4. Indexing

A separate index is built for each sub-collection. The indexing process parses the document, creates a `PageInfo` object to represent that document and adds it to the object store. The main fields of `PageInfo` are the following:

url:	extracted from the document
docno:	the TREC document number
description:	retrieved from Dmoz table (if available)
global ID:	retrieved from URL table
title:	title from the document
text:	content of document
citations:	text of all the citations referring to the document

Indexing required several preliminary steps: assigning unique doc IDs (avoiding duplicates), extracting page descriptions and categories from Dmoz, collecting links and anchor texts. Citations tables where built in parallel on 23 server blades: the process took 3.5 days. This includes the time to decompress documents and parse them: in a production system this step will be performed during crawling

The index is stored as compressed tables on disk that can be directly mapped to memory during search. Posting List compression uses variable byte coding, which we called *eptacode*, since it uses 7 bits per byte and a continuation bit, and is similar to the *vbyte coding* in [8]. In our experiments *eptacode* produced 23% smaller postings than those achieved with techniques like local Bernoulli with Golomb encoding as reported in [12]. Posting lists contain both position information and *color* tagging, that can be used to specify properties of text, like occurrence within specific zones of the document (title, anchor, heading), lexical features like size, capitalization or semantic tags like part-of-speech, name-entity tags, etc.

Documents are described as objects with several fields extracted from various sources and stored in the IXE object store. The indexer uses a specific document reader for HTML, which extracts additional fields like title and description.

Tokenization by the document reader assigns a different token type ('color') to each term that is used for weighting. These colors include:

1. Textual tokens: extracted from the document's text and divided into:
   a) Title – extracted from the document title
   b) Heading – extracted from the document's headers
   c) Anchor – extracted from the document's anchor elements
   d) Regular tokens – all the rest
2. Anchor tokens: extracted from the anchors of document's in-links

3. Description tokens: extracted from the description of the document, either from Dmoz or from the META description tag
4. URL tokens: extracted from the document's URL.

Indexing was performed in parallel on all blades, building 23 shards, each containing ~1 million documents, with an index size of 4.2 GB, and an additional 4.3 GB for a compressed copy of the documents. A document cache contains compressed copies of all the original documents.

Overall the index size is 92 GB, made up of three parts as listed in Table 1. Indexing the whole collection 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 we 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 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.

1.5. Search

The retrieval score computed by the search engine combines the following factors:

1. classical cosine measure based on *tf* and *idf*
2. term color weighting, i.e. text occurrence in title, heading, anchor within the document, occurrence in url or description (either from an HTML meta tag or from Dmoz listing) and occurrence in the anchor of a referring document
3. a proximity boost for terms appearing closer within a document
4. number of incoming links
5. ratio of non content links (IMG, INPUT, etc.) to document length
6. URL path length.

The score for a page p was computed as in [1] according to the number n of links pointing to p, by the formula:

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

where N is an upper bound on a page's in-link number.

The content rate was computed as follows:

$$Cr = \frac{|p|}{1 + 4o + 10z}$$

where $|p|$ is the length of the document in words, o is the number of out-links and z is the number of non-content tags.

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.

For instance a query for the words "oil" and "industry", asking for the first 10 results is expressed as:

```
?q=oil+industry&start=0&num=10
```

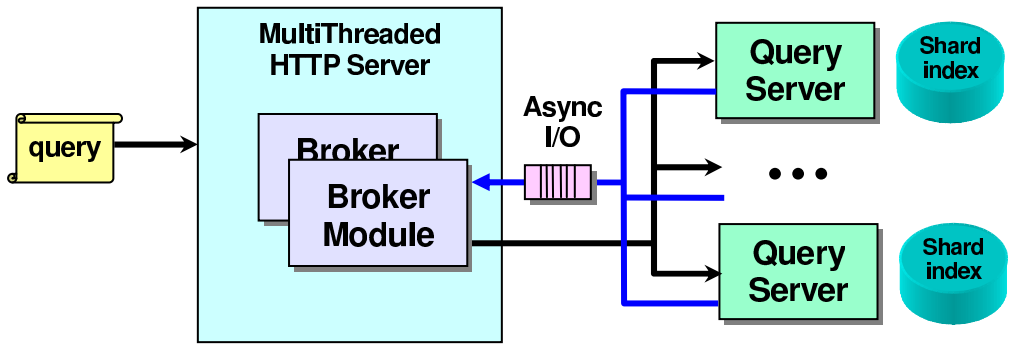
where `q` is the parameter for the query itself, `num` is the parameter for the number of results and `start` the parameter for the position of first result requested.

In the case of Web Service, responses are sent back as SOAP messages. The binary communication protocol is used for instance by a query broker that collects results from other query servers. The binary protocol exploits the serialization of C++ objects provided by the reflection facility in IXE. This serialization is much more compact than using the SOAP protocol: for instance a single query result requires just 28 bytes, hence a query server can return over 50 results in a single Ethernet packet.

The search service exposes three methods:

1. `search(query, start, num)`: returns the top `start+num` ranking documents for the given query
2. `getDoc(ID)`: returns the document identified by the given ID
3. `getDocInfo(ID, terms)`: returns the `PageInfo` representing the document, including the snippets containing the given terms

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.



2. Performance

Search in IXE exploits the Small Adaptive Set Intersection algorithm [5]: posting lists are sorted by *docid*, stored compressed and use skip lists to optimize the handling of frequent words (for the GOV2 collection we indexed without removing stop words). A separate index file contains position information for handling proximity queries: this file is also mapped to memory during search, but only piecewise, since it may be larger than the 4 GB limit on 32-bit architectures.

Search is performed in parallel: each blade runs a search service on its shard; a broker collects and sorts the results. Snippet extraction is also distributed.

A search for 10,000 results for a single proximity query on the whole collection takes an average of 0.3 sec.

We compared the performance of IXE with two other systems, Zettair [11] a search engine written in C by the Search Engine Group at RMIT University, and Lucene [8] a search engine written entirely in Java. The benchmarks were run on a single 800 MHz Pentium III, with 1 GB of memory, on the directory GX000 of the GOV2 collection, which consists of 89,771 documents with 719,670 distinct terms. Index sizes were 207 MB for IXE, 183 MB for Zettair and 332 MB for Lucene. Indexing times were 13 min. for IXE, 6 min. for Zettair and 4 hours for Lucene.

Since IXE performs proximity calculations also for AND queries in order to boost rank, the comparison was done on phrase searches.

Query	Results	IXE	Zettair	Lucene
“click here”	4932	10	15	37
“personal information”	901	12	32	37
“united states”	14052	17	39	57
“site map”	14492	25	41	88
“privacy policy”	13418	27	39	55
“contact us”	19311	36	68	112
“home page”	14668	46	57	141

Times are in milliseconds and were obtained after repeating the query twice, in order to allow for the effects of memory caching.

For queries with large number of results, the use of skip lists by IXE proves indeed effective.

3. TeraByte track

For ad-hoc retrieval, it is generally accepted that an approach based on the vector space model, combined with suitable ranking formula (e.g. BM25), is suitable to achieve high precision scores. The benefits of using proximity operators or phrase matching are instead controversial since their use has rarely shown substantial improvements of retrieval results. On the other hand, these methods dominate Web search: Boolean AND queries are the default and proximity is accounted in ranking of results. This may be explained by the peculiar requirements for Web search: the expected response time even on a huge collection is very short, hence it is impossible to examining all possible results of a disjunctive query; queries contain very few keywords, hence other criteria must be exploited to guess relevance.

We wished to assess the effectiveness of the techniques used by Web search engines in the TeraByte task, including in particular conjunctive queries and scoring exploiting: link popularity, proximity boost, home page score, descriptions and anchor text.

Since we expected conjunctive queries to be subject to lower recall than disjunctive queries, we explored a new method for query expansion, based on clustering snippets.

3.1. The baseline approach

In the experiments we tried to evaluate the effectiveness of this approach to query expansion and rewriting using phrase and proximity queries on retrieval from a large collection.

We built queries starting from the title of each topic and extracting name entities and noun phrases. For example for topic 701 (oil industry history) the phrase "*oil industry*" and the term "*history*" are extracted. These elements are combined to form a proximity query like this:

```
text matches proximity 20 [ 'oil industry' history ]
```

Simple stemming was applied to add as an alternative in the query the singular form of plural terms. This processing produced an overall list of 114 queries for the 50 topics.

3.2. Query expansion through Web snippet clustering

The queries generated in the first step sometimes retrieve too few results, since topics titles are short, sometimes too many, since they include terms with broad meaning (history, issue, example). More effective queries can be produced using techniques of query expansion.

Relevance feedback can be used as for query modification in two ways: expanding the query by adding new terms from relevant documents [15]; adjusting the individual term weights based on the user relevance judgments [14]. Query expansion is more appropriate to our goal of increasing recall.

Query expansion through *local clustering*, as discussed in [2], expands the query with terms correlated to the query terms, selected from clusters built from the documents retrieved for the original query. Retrieving and clustering all or even the top hundred ranking documents is considered too expensive.

Query expansion through *local context analysis* is based on the use of noun groups, instead of single words, as document concepts. Concepts for query expansion are selected from the top ranked documents based on their co-occurrence within passages in these documents.

Our approach is similar to local context analysis, except that we use a global collection (the Web), we use clustering of result snippets to select new query terms and when appropriate such terms are used as replacements for query terms rather than additions.

To generate new queries for a topic we performed two steps:

- finding new terms related to the topic (e.g. *petroleum*),
- combining these terms into meaningful queries for a topic (e.g. substituting "*petroleum*" for "*oil*", while keeping the other terms "*industry history*").

For finding additional terms for query expansion, we exploited clustering on an enriched collection, in this case the whole Web collection. We submitted the queries to the Vivisimo clustering engine [10] and we extracted the labels for the first level clusters (called "candidate

labels") from the response; for example, for topic 701 the query consisted of the terms *"oil industry" AND history* and produced the following labels: *Petroleum*, *Oil and Gas*, *Texas*, *Books*, *Market*, *Standard Oil*, *Collection*, *Drake*, *Iraq*, *Profiles*.

In order to select suitable labels for query expansion, we used an intuitive heuristics. A further query was submitted to Vivisimo for each of the candidate labels: if it produced clusters whose labels contained a term present in the original topic, then the label was considered suitable for query expansion, otherwise it was discarded. In the above example, labels like "*Market*" and "*Books*" were considered unrelated and hence discarded.

The labels deemed related were used to generate expanded queries as follows: terms in a candidate label replace the terms that appeared in the matching cluster label. For example, the query for the candidate label "*Petroleum*" produces a cluster with the term "*oil*" in its label, so the query "*petroleum industry" AND history* is obtained by replacing "*oil*" with "*Petroleum*" in the original query.

Through this process a total of 210 related labels were selected, generating additional query terms that were used to produce two (or more) variants of the original queries.

3.3. Merging query results

For each query in a topic up to 10,000 query results were retrieved and merged in a single list for the topic. Query results are a list of document IDs ordered by their retrieval score computed as described above. Instead of simply merging these lists and selecting the top ranking 10,000 documents, we classify queries into four categories:

- Original proximity queries: the ones directly extracted from topics, with a strict proximity constraint.
- Original non-proximity queries: the ones directly extracted from topics, with a loose proximity constraint.
- Generated proximity queries: the ones generated using clustering, with a strict proximity constraint.
- Generated non-proximity queries: the ones generated using clustering, with a loose proximity constraint.

Results obtained from queries in each category are expected to be more relevant than those from later categories. To take this into account, the overall score used in merging results from all queries involves a weight that decreases from the first to the last category. Weights have been assigned manually.

When a document is retrieved through more than one query, it gets the maximum of its scores.

3.4. Screening Results by means of Clustering

Identifying the subject of a document may be useful to sift them, distinguishing those most relevant to a topic. For example, for topic 702 (*pearl farming*), a number of retrieved documents were about child labor, and *pearl farming* was cited in passing among the sectors where child labor is exploited.

We experimented with the use of clustering to group documents and to screen them according to which cluster they belong to, lowering their rank if they appear in a cluster with little relation to

the topic, raising their rank if the cluster had a title that appeared relevant to the topic being searched.

We extracted from each document in the result list a few snippets containing the *topic query terms* (those terms used in retrieving the documents for a given topic). For example, for topic 701 (US oil industry history), the topic query terms included also *Drake*, obtained from query expansion, and the following snippets were extracted from one of the result documents:

> Mr. Peterson speaks about the need for Energy Independence during an Energy Town Hall at **Drake**'s Well... In August 1859, Colonel Edwin L. **Drake** completed the world's first successful **oil** well near Titusville and changed the course of **history**

The snippets from the results of each topic are passed to the clustering engine. Resulting clusters are processed similarly as it was done for queries generation: if a cluster label contains at least one topic query term, then the score for each document in that cluster is increased by the following factor:

$$log\ (resultsSize\ /\ (resultsSize - clusterSize))$$

where *resultsSize* is the number of results retrieved for the topic and *clusterSize* is the size of the cluster. This boosting is not applied to small clusters (those containing less than 10% of the total results). Clusters whose label does not match any topic query term are instead considered unrelated and the score of their documents is decreased by a 0.01 factor.

3.5. Combining results

We submitted four runs to the TREC 2004 TeraByte track.

The first run is our baseline run and was produced using a set of 552 queries: 114 extracted from the TeraByte topics title and 438 produced through query expansion by means of clustering. The latter set contained two variants for each query: one with a proximity 20 constraint and a 0.8 weight, and one without proximity and a 0.4 weight. The queries produced a total of 488,866 results, reduced to 225,445 after duplicate removal; no boosting was applied. This run took 161 seconds to complete, with an average time of 0.3 second per query (3.2 seconds per topic).

For the second run we reindexed the collection, adding a count of the number of informative and non-informative links in a page, which was used to lower the rank of documents having little contents: in particular pages describing single items like book library cards.

The queries were changed by adding non-proximity versions for queries that had returned few results, in an attempt to increase recall. This run used a set of 669 queries that produced 1,352,910 results, 275,741 after merge. This run took 296 seconds to complete, with an average time of 0.4 second per query (5.9 seconds per topic).

In the third run non-proximity queries were replaced with proximity queries with a wider proximity limit (from 20 to 200), to discard documents that contained the query terms in unrelated sections. The run consisted of 682 queries that produced 428,208 results, 148,156 after merging. This run took 190 seconds to complete, with an average time of 0.3 second per query (3.8 seconds per topic).

The fourth run was devised to measure the effects of relevance screening by clustering. The screening step was applied to the results obtained from the previous run. Extracting the snippets for all the results is more time consuming than processing the queries and takes about 40 minutes

overall. This operation consisted in sending the request for snippet, composed of the document ID and the terms to be used for snippet extraction, to the server that holds the document. The server retrieves the document from its local cache, uncompresses it, extracts the snippet and sends it back. Snippets for each topic are then processed by the clustering engine to produce clusters. The time required to produce the clusters varies from a few seconds for smaller result sets up to 20 minutes for the largest clusters of 10,000 documents. After this, calculating the boost factors and merge sorting the results required about 12 minutes.

4. Results

For comparison purposes we report the result of two unofficial runs: one that uses just topic title words and Okapi BM25 ranking, without the additional weights for link popularity and content rate; and a second one that uses conjunctions of the topic title words as queries, also without additional weights and without the boost for proximity that IXE normally applies to conjunctive queries.

Run	MAP	P@10	R-precision	Relevant
title only, disjunction (BM25)	0.16	0.46	0.25	4133
title only, conjunction (BM25)	0.14	0.38	0.22	4041
pisa1	0.05	0.25	0.09	4968
pisa2	0.09	0.33	0.16	6112
pisa3	0.10	0.37	0.17	5525
pisa4	0.10	0.38	0.16	5525

The first runs achieves a better than median score, while using conjunctive queries produces a 18% drop in precision.

The drop is more drastic in run pisa1, which uses proximity queries instead of conjunctions. Only relaxing the proximity constraints, in run pisa3, precision raises back to the level of simple conjunctions.

As expected, query expansion has a significant effect on recall, with an overall increase of over 50%.

The marginal improvements in run pisa4 lead us to conclude that the effects of screening by clustering are not worth the effort.

5. Conclusions

We built a distributed search engine that applies the strategies typically used by Web Search engines (conjunctive queries and ranking based on a combination of criteria) and compared its effectives to the techniques typically used for ad-hoc retrieval. Contrary to expectations, certain criteria, like boosting the rank for documents where query terms appear closer, have a negative effect on precision. We explored a new approach to query expansion, based on extracting terms from the clustering of snippets returned from the first query. The approach is effective and increases precision significantly and recall even more.

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] E. Amitay, et al.. Juru at TREC 2003 - Topic Distillation using Query-Sensitive Tuning and Cohesiveness Filtering, *Text REtrieval Conference (TREC) 2003 Proceedings*, 2004.

[2] R. Attar, A. S. Fraenkel. Local feedback in full-text retrieval systems. *Journal of the ACM*, 24(3):397-4117, 1977.

[3] G. Attardi, A. Cisternino. Reflection support by means of template metaprogramming, *Proceedings of Third International Conference on Generative and Component-Based Software Engineering*, *LNCS* 2186, 178-187, Springer-Verlag, Berlin, September 2001.

[4] C.-H. Chang, C.-C. Hsu, Enabling Web Information Retrieval through Query Expansion via Contrast Analysis, In *Proc. of the seventh International Conference on World Wide Web (WWW7)*, Brisbane, Queensland, Australia, 1998.

[5] 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.

[6] Dmoz, http://www.dmoz.org.

[7] E. Gabrilovich, A. Gontmakher. Heap Ltd., *Dr Dobb's Journal*, June 2003. http://www.cs.technion.ac.il/~gabr/papers/limited_heap.pdf

[8] Lucene, http://jakarta.apache.org/lucene/.

[9] F. Scholer, H.E. Williams, J. Yiannis, and J. Zobel. Compression of Inverted Indexes For Fast Query Evaluation, In K. Jarvelin and M. Beaulieu and R. Baeza- Yates and S. H. Myaeng, *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval*, Tampere, Finland, 222-229, 2002.

[10] Vivisimo, http://vivisimo.com.

[11] H. Williams, et al. The Zettair Search Engine, http://www.seg.rmit.edu.au/zettair.

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

[13] J. Xu, W. B. Croft. Query expansion using local and global document analysis. *Proc. ACM-SIGIR Conference on Research and Development in Information Retrieval*, Zurich, Switseraland, 4-11, 1996.

[14] S.E. Robertson, K. Sparck Jones. Relevance Weighting of Search Terms, *Journal of the American Society for Information Science*, 27(3):129-146, 1976.

[15] J.J. Rocchio. Relevance Feedback in Information Retrieval, in Salton G. (Ed.), *The SMART Retrieval System*, Englewood Cliffs, N.J.: Prentice-Hall, Inc., 313-323, 1971.

[16] G. Salton and C. Buckley. Improving retrieval performance by relevance feedback. Journal of the American Society for Information Science, 41(4):288-297, 1990.

[17] Franz, McCarley and Ward. Ad hoc, Cross-language and Spoken Document Information Retrieval at IBM, http://trec.nist.gov/pubs/trec8/papers/t8_ibm_hlt.pdf.