Load-Balancing and Caching for Collection Selection Architectures

Diego Puppin, Fabrizio Silvestri, Raffaele Perego
ISTI-CNR, Pisa
name.last@isti.cnr.it

Ricardo Baeza-Yates
Yahoo! Research, Barcelona/Santiago
ricardo@baeza.cl

Abstract— **To address the rapid growth of the Internet, modern Web search engines have to adopt distributed organizations, where the collection of indexed documents is partitioned among several servers, and query answering is performed as a parallel and distributed task. Collection selection can be a way to reduce the overall computing load, by finding a trade-off between the quality of results retrieved and the cost of solving queries. In this paper, we analyze the relationship between the collection selection strategy, the effect on load balancing and on the caching subsystem, by exploring the design-space of a distributed search engine based on collection selection. In particular, we propose a strategy to perform collection selection in a load-driven way, and a novel caching policy able to incrementally refine the effectiveness of the results returned for each subsequent cache hit. The combination of load-driven collection selection and incremental caching strategies allows our system to retrieve two thirds of the top-ranked results returned by a baseline centralized index, with only one fifth of the computing workload.**

I. Introduction

The Web has brought a dramatic change in the way people publish, collect and search information. Users rely more and more on search engine for any information task, often looking for their needs directly in the result page (the so-called *information snacking*[19]). This is why successful search engines have to index and search quickly billions of pages, for millions of users every day.

Very sophisticated techniques are needed to implement efficient search strategies for very large document bases. Parallel and distributed information retrieval systems are the most natural way to tackle this problem [2].

A typical distributed information retrieval system is organized into several layers [3]. Usually a multi-site system is exploited, where each site holds a copy (or portion) of the collection, and comprises a number of query servers. Each query server is, in turn, a hierarchy of modules structured as in Figure 1. Each module is responsible for a specific task, and it is fine-tuned to work in the best possible way given its peculiar workload.

To increase the overall throughput, each module caches different types of data. At the broker level, we can have a cache of query results, i.e. results of the most frequently submitted queries are cached for successive submissions of the same query. At the level of the search servers (IR cores in Figure 1), the system may instead cache the posting lists of the most frequently accessed terms, in order to reduce retrieval time. A combination of both can be used to achieve the best performance (as shown in [24]).

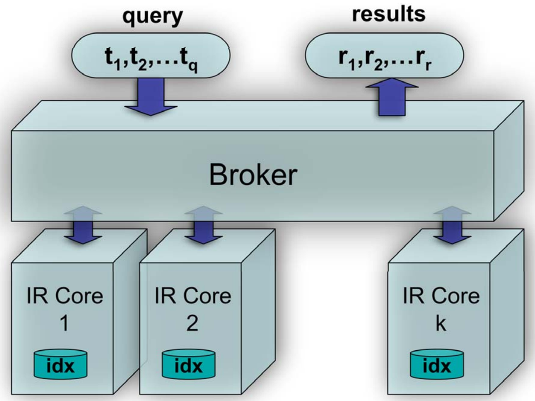

Fig. 1. Organization of a parallel information retrieval system.

Moreover, the broker is responsible for exploiting the partitioning policy of the document database. Depending on the type of data partitioning adopted, the broker responsibilities range from forwarding each query to all the IR cores (i.e. a pure *document-partitioned* IR system), to performing the selection of proper servers to which forward distinct query terms (i.e. a pure *term-partitioned* IR system). In all the cases the broker has to receive from the IR cores their partial results, and merge them to compute the final ranked list returned to the user.

In the past, a lot of different architectures have been proposed to enhance the scalability of distributed IR systems. A common way to tackle the distributed nature of data is *collection selection*. In this approach, a query is forwarded only to the servers considered to be the most authoritative for it. The way to measure the authoritativeness of a server is still an open problem, addressed by many different research groups.

In this paper, we present the problem from a novel perspective. We consider the realistic hypothesis that a Web IR system with collection selection is using a cache, and we want to exploit it to enhance not only the efficiency of the system, but also its effectiveness. To this end we introduce a strategy to guide collection selection using the instant load of the system to achieve a good load balance. The broker will poll more

INFOSCALE 2007, June 6-8, Suzhou, China
Copyright © 2007 ICST 978-1-59593-757-5
DOI 10.4108/infoscale.2007.892

and more servers, if they are idle, but will reduce its requests to only very relevant sub-collections if the system is heavily loaded.

We also discuss a novel class of caching strategies that we call *incremental caching*. When a query is submitted to our system, its results are looked for in the cache. In the case of a hit, some results previously retrieved from a subset of servers will be available from the cache. The incremental cache however will try to poll more servers for each subsequent hit, and will update the list of top-scoring results stored in the cache. Over time, the cached queries will get perfect coverage, because results from all the servers will be available. This is true, in particular, for common or bursty queries: the system will show great performance in answering them. On the other hand, the quality of results stored in the cache introduces another dimension which has to be considered when the cache is full, and one of its entry should be evicted to make place for a new query.

The rest of the paper is organized as follows. Section II discusses related work. Then, we present our collection selection model in Section III. Section IV discusses our strategies for load balancing and load reduction, while Section V introduces the class of incremental caching policies and proposes a framework for evaluating the effectiveness of its integration within a collection selection IR system. Section VI describes in detail the different caching policies possible in our framework. The experimental results are discussed in Section VII. Finally, Section VIII draws some conclusions and future work.

II. Related Work

There have been a lot of papers dealing with models for partitioning a document collection. Document partitioning (DP), in fact, has been shown by many researchers to be the best choice among parallelization schemes, by offering the best tradeoff between load balancing and load distribution [16], [7], [1].

Today, it is commonly held that realistic parallel IR systems will have to manage distinct indexes. In our opinion, a possible way to ensure timely and economic retrieval is designing a broker module so that it will forward a given query only to workers managing documents related to the query topic. In other words, we are speaking about adopting *collection selection* techniques aimed at reducing a query search space by querying only a subset of the entire group of workers available. Nevertheless, particular attention should be paid in using this technique. In fact, it could result in a loss of relevant documents, causing a degradation in effectiveness.

The most common approaches to distributed IR exploit a number of heterogeneous collections grouped by source and time period. A *collection selection index (CSI)*, summarizing each collection as a whole, is used to decide which collections are most likely to contain relevant documents for the query. Document retrieval will take place at these collections only. In [12] and [9] several selection methods have been compared. The authors showed that the naïve method of using only a collection selection index lacks in effectiveness. Many proposals tried to improve both the effectiveness and the efficiency of the previous schema.

Moffat *et al.* [18] use a centralized index on blocks of B documents. For example, each block might be obtained by concatenating documents. A query first retrieves block identifiers from the centralized index, then searches the highly ranked blocks to retrieve single documents. This approach works well for small collections, but causes a significant decrease in precision and recall when large collections have to be searched.

In [8], the authors compare the retrieval effectiveness of searching a set of distributed collections with that of searching a centralized one. The system they use to rank collections is an inference network in which leaves represent document collections, and representation nodes represent the terms that occur in the collection. The probabilities that flow along the arcs can be based upon statistics that are analogous to tf and idf in classical document retrieval: document frequency df (the number of documents containing the term) and inverse collection frequency icf (the number of collections containing the term). They call this type of inference network a *collection retrieval inference network*, or *CORI* for short. They found no significant differences in retrieval performance between distributed and centralized searching when about half the collections, on average, were searched for a query.

ReDDE, presented in [25], is an improvement over CORI for collections that are uncooperative or of different sizes. It also improves on the strategy to re-rank the results coming from those collections. While very effective, it does not apply to our case, as collections are cooperative and balanced (similar size).

In [29], collection selection strategies using cluster-based language models have been investigated. Xu *et al.* proposed three new methods of organizing a distributed retrieval system based on the basic ideas presented before. This three methods are *global clustering*, *local clustering*, and *multiple-topic representation*. In the first method, assuming that all documents are available in one central repository, a clustering of the collection is created; each cluster is a separate collection that contains only one topic. Selecting the right collections for a query is the same as selecting the right topics for the query. This method is appropriate for searching very large corpora, where the collection size can be in the order of terabytes. The next method is *local clustering* and it is very close to the previous one, except the assumption of a central repository of documents. This method can provide competitive distributed retrieval without assuming full cooperation among the subsystems. The disadvantage is that its performance is slightly worse than that of global clustering. The last method is *multiple-topic representation*. In addition to the constraints in local clustering, the authors assume that subsystems do not want to physically partition their documents into several collections. A possible reason is that a subsystem has already created a single index and wants to avoid the cost of reindexing. However, each subsystem is willing to cluster its documents and summarize its collection as a number of topic models for effective collection selection. With this method a collection corresponds to several topics. Collection selection is based on how well the best topic in a collection matches a query. The advantage of this approach is that it assumes minimum cooperation from the subsystem. The disadvantage is that it is less effective than both global and local clustering. This paper actually introduces a technique that is similar to ours, but while the authors base their distribution schema only on knowledge coming from the textual collection, we also rely on knowledge coming from the past usage of the system.

Caching is a useful technique on the Web: it enables a shorter average response time, it reduces the workload on back-end servers and the overall amount of utilized bandwidth. When a user is browsing the Web, both his/her client and the contacted servers can cache items. Browsers can cache Web objects instead of retrieving them repeatedly, while servers can cache pre-computed answers or partial data used in the computation of new answers. A third possibility, although of less interest to the scope of this paper, is storing frequently requested objects in the proxies to mediate the communication between clients and servers [20].

Query logs constitute a valuable source of information for evaluating the effectiveness of caching systems. While there are several papers analyzing query logs for different purposes, just a few consider caching for search engines. As noted by Xie and O'Hallaron [27], many popular queries are in fact shared by different users. This level of sharing justifies the choice of a server-side caching system for Web search engines.

Previous studies on query logs demonstrate that the majority of users visit only the first page of results, and that many sessions end after just the first query [26], [14], [13], [4].

In [26], Silverstein *et al.* analyzed a large query log of the AltaVista search engine, containing about a billion queries submitted in more than a month. Tests conducted included the analysis of the query sessions for each user, and of the correlations among the terms of the queries. Similarly to other works, their results show that the majority of the users (in this case about 85%) visit the first result page only. They also show that 77% of the users' sessions end up just after the first query. A similar analysis is carried out in [14]. With results similar to the previous study, they concluded that while IR systems and Web search engines are similar in their features, users of the latter are very different from users of IR systems. A very thorough analysis of users' behavior with search engines is presented in [13]. Besides the classical analysis of the distribution of page-views, number of terms, number of queries, etc., they show a topical classification of the submitted queries that point out how users interact with their preferred search engine. Beitzel *et al.* [4] analyzed a very large Web query log containing queries submitted by a population of tens of millions users searching the Web through AOL. They partitioned the query log into groups of queries submitted in different hours of the day. The analysis, then, tried to highlight the changes in popularity and uniqueness of topically categorized queries within the different groups.

One of the first papers that exploits user query history is by Raghavan and Sever [22]. Although their technique is not properly caching, they suggest using a *query base*, built upon a set of persistent "optimal" queries submitted in the past, in order to improve the retrieval effectiveness for similar future queries. Markatos [17] shows the existence of temporal locality in queries, and compares the performance of different caching policies.

Lempel and Moran propose *PDC* (Probabilistic Driven Caching), a new query answers caching policy based on the idea of associating a probability distribution with all the possible queries that can be submitted to a search engine [15]. PDC uses a combination of a SLRU cache [17] (for requests of the first page) and a heap for storing answers of queries requesting pages next to the first. Priorities are computed on previously submitted queries. The distribution is built over statistics computed on the previously submitted queries. For all the queries that have not previously seen, the distribution function evaluates to zero. This probability distribution is used to compute a priority value that is exploited to order the entries of the cache: highly probable queries are highly ranked, and have a low probability to be evicted from the cache. Indeed, a replacement policy based on this probability distribution is only used for queries regarding pages subsequent to the first one. For queries regarding the first page of results, a *SLRU* policy is used [17]. Furthermore, *PDC* is the first policy to adopt prefetching to anticipate user requests. To this purpose, *PDC* exploits a model of user behavior. A user session starts with a query for the first page of results, and can proceed with one or more *follow-up* queries (i.e., queries requesting successive page of results). When no follow-up queries are received within τ seconds, the session is considered finished. This model is exploited in *PDC* by demoting the priorities of the entries of the cache referring to queries submitted more than τ seconds ago. To keep track of query priorities, a priority queue is used. *PDC* results measured on a query log of AltaVista were very promising (up to 53.5% of hit-ratio with a cache of $256,000$ elements and 10 pages prefetched).

Fagni *et al.* show that combining static and dynamic caching policies together with an adaptive prefetching policy achieves a high hit ratio [11]. In their experiments, they observe that devoting a large fraction of entries to static caching along with prefetching obtains the best hit ratio. They also show the impact of having a static portion of the cache on a multithreaded caching system. Through a simulation of the caching operations they show that, due to the lower contention, the throughput of the caching system can be doubled by statically fixing a half of the cache entries.

The work presented in this paper tries to combine the previous two approaches by designing a new family of caching policies, explicitly thought for a distributed IR system based on collection selection.

Query/Doc	d1	d2	d3	d4	d5	d6	...	dn
q1	-	0.5	0.8	0.4	-	0.1	...	-
q2	0.3	-	0.2	-	-	-	...	0.1
q3	-	-	-	-	-	-	...	-
q4	-	0.4	-	0.2	-	0.5	...	0.3
...	...	...	...	...	...	...	...	...
qm	0.1	0.5	0.8	-	-	-	...	-

TABLE I

IN THE QUERY-VECTOR MODEL, EVERY DOCUMENT IS REPRESENTED BY THE QUERY IT MATCHES (WEIGHTED WITH THE SCORE).

PCAP	dc1	dc2	dc3	dc4	dc5	$r_q(qc_i)$
qc1		0.5	0.8	0.1		0.2
qc2	0.3		0.2		0.1	0.8
qc3	0.1	0.5	0.8			0

$$\begin{array}{lcrcrcrcl}
r_q(dc_1) & = & 0 \times 0.2 & + & 0.3 \times 0.8 & + & 0.1 \times 0 & = & 0.24 \\
r_q(dc_2) & = & 0.5 \times 0.2 & + & 0 & + & 0 & = & 0.10 \\
r_q(dc_3) & = & 0.8 \times 0.2 & + & 0.2 \times 0.8 & + & 0 & = & 0.32 \\
r_q(dc_4) & = & 0.1 \times 0.2 & + & 0 & + & 0 & = & 0.02 \\
r_q(dc_5) & = & 0 & + & 0.1 \times 0.8 & + & 0 & = & 0.08
\end{array}$$

TABLE II

EXAMPLE OF PCAP TO PERFORM COLLECTION SELECTION.

III. COLLECTION SELECTION USING THE QV DOCUMENT MODEL

Traditionally, two main ways are used to model the documents from a collection: the so-called *bag-of-words* model, which represent a document with the set of terms present in it; and the *vector space*, which gives a different weight to every occurring term.

These two representations are very long boolean or real vectors, and even though they have been successfully used in traditional IR systems, they are prone to the curse of dimensionality [5] when used in other contexts like clustering [30].

The QV representation of a document is built out of a query-log. A reference search engine is used in the first phase: for every query in the training set, the system stores the top results, along with their score. In our case, we store 100 results.

Table I gives an example. The first query $q1$ recalls, in order, $d3$ with score 0.8, $d2$ with score 0.5 and so on. Query $q2$ recalls $d1$ with score 0.3, $d3$ with score 0.2 and so on. We may have empty columns, when a document is never recalled by any query (in this example, $d5$). Also, we can have empty rows when a query returns no results ($q3$).

To be more precise, let Q be a query log containing queries $q_1, q_2, \ldots, q_m$. Let $d_{i1}, d_{i2}, \ldots, d_{in_i}$ be the list of documents returned as results to query q_i. Furthermore, let r_{ij} be the score that document d_j gets as result of query q_i (0 if the document is not a match).

A document d_j is represented as an m-dimensional vector $\bar{d_j} = [\bar{r}_{ij}]^T$, where $\bar{r}_{ij} \in [0, 1]$ is the normalized value of r_{ij}:

$$\bar{r}_{ij} = \frac{r_{ij}}{\sum_{ij} r_{ij}}$$

We call this a *query vector*, because we represent documents by the score of the queries it matches. The $\bar{r}_{ij}$ values form a contingency matrix R, which can be seen as a joint probability distribution and used by the co-clustering algorithm by Dhillon *et al.* [10]. Their approach creates, simultaneously, clusters of rows (queries) and columns (documents) out of an initial matrix, with the goal of minimizing the loss of information. To improve performance, empty columns and rows are removed from the matrix before clustering. Documents corresponding to empty columns are put together in an *overflow* document cluster.

The result of co-clustering is a matrix $\widehat{P}$ defined as:

$$\widehat{P}(qc_a, dc_b) = \sum_{i \in qc_b} \sum_{j \in dc_a} \bar{r}_{ij}$$

Each entry $\widehat{P}(qc_a, dc_b)$ sums the contributions of $\bar{r}_{ij}$ for the queries in the query cluster qc_a and the documents in document cluster dc_b. The values of this matrix, called PCAP, are important because they measure the relevance of a document cluster to a given query cluster.

This induces a simple but effective collection selection algorithm. The queries belonging to each query cluster are joined together into *query dictionary* files. Each dictionary files stores the text of each query belonging to a cluster, as a single text file. When a new query q is submitted to the IR system, the BM25 [23] metric is used to find which clusters are the best matches: each dictionary file is considered as a document, which is indexed in the vector space, and then queried with the usual BM25 technique. This way, each query cluster qc_i receives a score relative to the query q, say $r_q(qc_i)$.

This is used to weight the contribution of PCAP $\widehat{P}(i, j)$ for the document cluster dc_j, as follows:

$$r_q(dc_j) = \sum_i r_q(qc_i) \times \widehat{P}(i, j)$$

Table II gives an example. The top table shows the PCAP matrix for three query clusters and five document clusters. Suppose BM25 scores the query-clusters respectively 0.2, 0.8 and 0, for a given query q. After computing $r_q(dc_i)$, we will choose the collection $dc3$, $dc1$, $dc2$, $dc5$, $dc4$ in this order.

In [21], PCAP is shown to out-perform CORI for the task of combined document partitioning and collection selection. We will use PCAP in all our experiments including collection selection. In particular, the overflow cluster, with documents never recalled, is queried only when there is no match in the query dictionaries, or when the server holding it is idle.

IV. APPLICATIONS TO LOAD REDUCTION

Document partitioning is the strategy usually chosen by the most popular web search engines [6]. Each document in the base is assigned to one computing server, which creates a local index of its subset of documents. By sharing the global statistics on term frequencies, the servers can have coherent document scoring. A broker, in front of the computing servers, has the responsibility of routing the query to the servers, and collecting results from them.

The broker may choose among two main strategies for scheduling a query: a naïve, yet very common, way is to broadcast each query to all the underlying servers; the second one is to perform *collection selection* and use only a subset of servers. By doing so, the total computing load can be reduced. Here, we want to minimize the number of queried collections that are required to achieve a given level of precision. In [21], we show that using collection selection we can get very good coverage by querying only a limited number of clusters. We present here a set of strategies to reduce the computing load in a search engine, by exploiting the QV's strength in collection selection.

If the search engine broadcasts the query to each server, we have that each of them, obviously, has to elaborate 100% of the queries. If we choose to query only one of N servers (the most promising one), each server, on average, will elaborate a fraction $1/N$ of the queries. There can be peaks when one server is hit more often than the others, which can raise the fraction it has to elaborate in a given time window. Clearly, this strategy reduces the number of results returned to the user. There is a trade-off in result quality and computing load. If we want to increase the quality, we can query more servers each time, with a growth in the average load to each server.

Due to the nature of data used in our test set (web pages and a query log), we do not have a large enough collection of human-chosen relevant documents for each query: it is impossible to have a valid measure of *precision*. Thus, following the example of previous works [28], we count the *coverage*, i.e. how many of the top results, as returned by a reference search engine, are available from the selected collections (see the next section). In [21], we show that, in a configuration with 16+1 servers, the coverage is about one third, for different values of M.

We simulated a distributed search engine, doing some simplifying assumptions.

- The computing load of an IR core to answer one query on its local index is counted as 1, independently from the query and the queried server. This is justified by the fact that the sub-collections are of similar size. The server holding the overflow collection, composed of about 50% documents, is considered to be 16 times slower than the other servers.
- We measure the computing load of a server as the number of queries that are forwarded to it within a window of the query load. Unfortunately, we could not use information on query timing. So, a server contacted for each query in a given window, has 100% load. A server contacted by one query out of two, has 50% load. We imagine to have very big machines that can handle the full load, and then we cap the maximum load so to use smaller, low-fat machines.
- The search engine front-end can merge the results coming from the servers and sort them. This can be done just by computing the global term statistics at the end of the indexing.

Our simulation works as follows. Each query is submitted to a centralized search engine that indexes all the documents. We used Zettair[1], a compact and fast text search engine designed and written by the Search Engine Group at RMIT University.

Using the clustering results of the co-clustering algorithm, we know how many of the top-ranking documents are stored in each sub-collection.

We use our collection selection strategy to rank the document clusters, and then we choose to which servers we should broadcast the query, according to different strategies (below). At this point, we can record an increase in the load for the servers hit by the query, and we compute the coverage.

Collection selection was done in different ways:

- *Fixed* $< T >$: it chooses the first T servers, with T given once for all. The computing load can increase freely in this configuration. This was used to measure the computing power required to sustain a guaranteed number of queried servers.
- *Load-driven basic* $< l >$: the system contacts all servers, with different *priority*. The query will be forwarded first to the best server, unless it is overloaded. The second server is contacted with lower priority, i.e. if its load is not greater than, say, 90% the maximum. The third one is also contacted with even lower priority, i.e. in the case its load is lower than, say, 80% the maximum, and so on. The last one is contacted only if it is idle. This is done to *prioritize* queries to the most promising servers: if two queries involve a server s, and the load in s is high, only the query for which s is very promising will be served by s, and the other one will be dropped. This way, the overloaded server will be not hit by queries for which it is only a second choice.
  We set the load threshold l to match the values measured with the previous strategy. For instance, if we measure that with with *fixed 4*, the peak load is 40%, we compare it with *load-driven basic* $< 40\% >$.
- *Load-driven boost* $< l, T >$: same as load-driven, but here we contact the first T servers with maximum priority, and then the other ones with linearly decreasing priority. By boosting, we are able to keep the lower loaded servers closer to the load threshold. Boosting is valuable when the available load is higher, as it enables us to use the lower loaded servers more intensively.

The *fixed* strategy is used to measure the load needed to guarantee that T servers will be used to answer each query, and the coverage we can get under this scheme. The load-driven strategies are ways to exploit the difference in computing load of different servers.

The interaction with a caching system is very interesting. If we have a fixed selection, the cache will store the results from the chosen T servers. In case of a hit, the stored results will be used to answer the query.

[1] Available under a BSD-style license at http://www.seg.rmit.edu.au/zettair/.

In the case of the load-driven selection, the cache still stores the results of the servers that accepted the query. As just said, many servers can be dropped if they are overloaded or the priority is low. In case of a hit, the system will contact again the servers not contacted before. If they answer this time, their results will be added to the cache. This way, a repeated query can, over time, get very good coverage.

V. Incremental Caching

As shown in Figure 1, a distributed Web IR system is composed of a set of k IR core servers. Each of them usually indexes a disjoint sub-collection of documents, which are partitioned according to some predetermined strategy. In the following, we will give some definitions that will serve to define our family of *Incremental Caching* policies.

Definition 1: A *query-result record* is a quadruple of the form $< q, p, \overline{r}, \overline{s} >$, where: q is the query string, p is the number of the page of results requested, $\overline{r}$ is the ordered list of results associated to q and p, $\overline{s}$ is the set of servers from which results in $\overline{r}$ are returned. Each result is represented by a pair $< doc\_id, score >$.

As anticipated, if we cache an entry $< q, p, \overline{r}, \overline{s} >$, this means that only the servers in $\overline{s}$ were selected and polled, and that they returned $\overline{r}$. Also, since in an incremental cache results stored might be updated, we need to store the score (along with the document identifier) to compare the new results with the old ones.

From this first definition, we can see the first difference between a traditional and an incremental cache: results in an incremental cache are continuously modified by adding results from the servers that have not been queried yet. The set $\overline{s}$ serves to this purpose and keeps track of the servers that have been contacted so far.

Definition 2: The result list returned by the IR system (using caching and collection selection) for a query q is indicated as $r(q)$.

Definition 3: The result list obtained by querying all the k IR core servers for a query q is denoted by $r^*(q)$.

We can have that $r(q) \not\subset r^*(q)$, because the subset of IR core servers queried so far returned low-relevance results that are not included when we get additional relevant results.

Definition 4: Let q be a query, and R be an incremental replacement policy. We define *coverage* for a query the value

$$c_R = \frac{|r(q) \cap r^*(q)|}{|r^*(q)|}$$

If $|r^*(q)| = 0$, c_R is not defined and does not contribute to average.

Definition 5: Let Q be a query stream, and $Q_1 \subset Q$ the set of queries with at least one match. Let R be an incremental caching replacement policy. We define the average coverage as:

$$c_R(Q) = \frac{\sum\limits_{q \in Q_1} c_R(q)}{|Q_1|}$$

We are also interested in measuring the peak load that a server has to manage when dealing with a query stream. This is important because a server has to be sized up to the peak load, if we want to be able to serve all request.

Definition 6: For each IR core server in the system, given an incremental cache replacement policy R, a query stream Q, and a window size W, we call *instant load at time t* $l_R^{i,t}(Q)$ the fraction of queries answered by server i from the set of W queries ending at t. We define the *peak load for i* $l_R^i(Q)$ as the maximum $l_R^{i,t}(Q)$ over t, and the *maximum load* $\overline{l}_R(Q)$ as the maximum $l_R^i(Q)$ all over the k servers.

In our experiments, we set W equal to 1000, i.e. we keep track of the maximum fraction of queries that hit a server out of a rotating window of 1000 queries. With no cache and no collection selection, we expect to have a maximum load of 100%.

Definition 7: For a query stream Q, and an incremental caching policy R, the hit-ratio $h_R(Q)$ is the number of queries answered by the incremental cache divided by the number of queries in Q.

Now that we have defined these concepts, we can easily define the three parameters that will help us in measuring the performance of an incremental caching system.

Performance Indicators of an Incremental Cache
The performance of an incremental cache using replacement policy R on a query stream Q can be measured by three different metrics:

1) the *hit-ratio* $h_R(Q)$;
2) the *average coverage* $c_R(Q)$;
3) the *maximum load* $\overline{l}_R(Q)$.

VI. Different Types of Incremental Caches

We will present a list of possible cache approaches, starting from a very simple system to more complex incremental caches. With this, we would like to explore the design-space of this important component of an IR system.

A. No caching, no collection selection

This policy consists simply of forwarding each query to each IR core, with no caching at all.

```
$\Phi_N(q, p)$

1) Forward the query to all the $k$ servers;
2) collect results $\overline{r}$;
3) don't cache results;
4) return $\overline{r}$.
```

B. No caching, selection policy $\rho(q)$

This policy applies collection selection, in order to reduce the computing load, but does not perform caching. The system selects a subset $\overline{s}$ of the k IR core servers using the collection selection strategy $\rho(q)$, and then forwards them the query. The selection policy can be as simple as a random choice, or more complex as CORI [8] or PCAP [21].

```
$\Phi_\rho(q,p)$

  1) Select $\overline{s}$ using $\rho(q)$;
  2) forward the query to $\overline{s}$;
  3) collect results $\overline{r}$ from $\overline{s}$;
  4) don't cache results;
  5) return $\overline{r}$.
```

C. Caching policy P, no collection selection

This caching policy corresponds to apply a standard cache (using replacement policy P) to a pure document-partitioned distributed IR system. If the results for a query are not found in cache, then the query is forwarded to all the servers.

```
$P(q,p)$

  1) Look up ($q$,$p$).
  2) If found $< q, p, \overline{r} >$:
      a) update the cache according to $P$;
      b) return $\overline{r}$.
  3) If not found:
      a) forward the query to all servers;
      b) let $\overline{r}$ be the result set;
      c) select and remove the best candidate for
         replacement;
      d) store $< q, p, \overline{r} >$;
      e) return $\overline{r}$.
```

Please note that, the caching policy could choose *not* to remove any stored entry and drop the new results instead, if the entry is believed to be a one-timer. This is planned to be part of our future work.

D. Caching policy P, selection policy $\rho(q)$

In this case, the cache is storing the results coming from the selected servers, for the query q. Subsequent hits are not changing the result set.

```
$P_\rho(q,p)$

  1) Look up ($q$,$p$).
  2) If found $< q, p, \overline{r} >$:
      a) update the cache according to $P$;
      b) return $\overline{r}$.
  3) If not found:
      a) forward the query to a subset of the $k$
         servers, $\overline{s}$, selected as $\rho(q)$;
      b) let $\overline{r}$ be the result set;
      c) select and remove the best candidate for
         replacement;
      d) store $< q, p, \overline{r} >$;
      e) return $\overline{r}$.
```

Caching	Yes / No / Incremental result update
Collection Selection	CORI / PCAP / Random / None
Selection Strategy	Fixed number (choosing N) / Load driven (choosing load)
Cache partitioning	Full dynamic / SDC

TABLE III
THE DESIGN-SPACE FOR A COLLECTION SELECTION SYSTEM WITH CACHING.

E. Incremental caching policy $\widetilde{P}$, selection policy $\rho(q)$

This approach differs from the previous because, at each hit, the system adds results to the cached entry, by querying further servers each time.

```
$\widetilde{P}_\rho(q,p)$

  1) Look up ($q$,$p$).
  2) If found $< q, p, \overline{r}, \overline{s} >$:
      a) forward the query to the appropriate sub-
         set of servers (excluding $\overline{s}$); add them to
         $\overline{s}$;
      b) add the results to $\overline{r}$, and update $\overline{s}$;
      c) update the cache according to $\widetilde{P}$;
      d) return $\overline{r}$.
  3) If not found:
      a) forward the query to a subset of the $k$
         servers, $\overline{s}$, selected as $\rho(q)$;
      b) let $\overline{r}$ be the result set;
      c) select and remove the best candidate for
         replacement;
      d) store $< q, p, \overline{r}, \overline{s} >$;
      e) return $\overline{r}$.
```

Once chosen the main caching structure, several options are still available for the collection selection approach and the cache structure.

Load-driven collection selection: There are two main ways of performing collection selection, as shown in the previous section: *fixed* or *load-driven* (possibly with *boost*). In the second case, clearly, the broker has to model the dynamic load of each server, e.g. by counting the number of queries forwarded to each in a time window. The priority is used to prevent queries from overloading low-relevance servers, leaving them free to answer when the priority is higher.

Static-Dynamic Caching (SDC): The system could use the SDC caching policy introduced by Fagni [11], and reserve a part of the resources to a static cache.

All these choices can be combined in any desired combinations, generating a very large design-space (see Table III).

VII. Experimental Results

A. Setup

We performed all our test using the WBR99 collection. WBR99 consists of 5,939,061 documents, about 22 GB uncompressed, representing a snapshot of the Brazilian Web (domains .br) as spidered by www.todobr.com.br. It comprises about 2,700,000 different terms. We could use also the query log of www.todobr.com.br for the period January through October 2003. The main features of our data are:

d : 5,939,061 documents taking (uncompressed) 22 GB;
t : 2,700,000 unique terms;
t' : 74,767 unique terms in queries;
tq : 494,113 (190,057 unique) queries in the training set;
$q1$: 194,200 queries in the test set;
ed : 3,128,366 documents never recalled;
dc : 16 + 1 document clusters;
qc : 128 query clusters.

For our experiment, we used Zettair. We modified it so to implement the PCAP collection selection strategy. The query-vector representation is built using, as a training set, the first three weeks of our query log, which comprise about 190,000 unique queries. We chose to use only unique queries, as samples of the users' interests.

This means that, at the logical level, documents are represented by vectors in $\mathbb{R}^{190,000}$. On the other side, the vector-space representation is a vector in $\mathbb{R}^{2,700,000}$, because we have 2,700,000 distinct terms in our base.

After removing the empty QV documents (documents never recalled during training), the collection was clustered into 16 document clusters. The empty QV documents were grouped into an *overflow* cluster. This means that the overall cluster is holding about 50% of the collection, while the other 16 servers are holding 1/32 each. In our experiments, we considered the overflow cluster to be 16 times slower.

We also created 128 query clusters, which were transformed into query dictionaries and indexed to be used by PCAP (as explained above). In the test we show below, co-clustering algorithm was run with 20 iterations. Due to limited space, we cannot show the results with different configurations, but they are very similar in the main trends.

We used the fourth week from the query-log as test set: the queries from the fourth week were submitted to the distributed search engine, and we measured the coverage of the selected collections. In this case, we do not use unique queries, but all occurring queries, because this is a more faithful estimation of the coverage as perceived by users.

We do not show here the results measured when using the fifth week, because they do not present major differences.

B. System configurations and results

We tested five system configurations, with three cache sizes (8000, 16000 and 32000 entries):

1) No caching, no collection selection;
2) Caching policy LRU, no collection selection;
3) Caching policy LRU, selection policy PCAP, fixed choice (1, 2, 4, 8);
4) Caching policy LRU, selection policy PCAP, load-driven choice (21.1, 32.5, 43.9, 55.5);
5) Incremental caching policy LRU, selection policy PCAP, load-driven choice (21.1, 32.5, 43.9, 55.5).

The values 21.1, 32.5, 43.9 and 55.5 represent the peak load reached in the third configuration. In configuration 4 e 5, we try to poll servers, in a load-driven way, up to these load thresholds. Load-driven basic selection resulted inferior to load-driven boosted selection in all circumstances. Due to limited space, we do not show the comparison, and in the following we will always use the boosted version. Results are shown in Table IV.

The first experiment is used to set up our reference. It reaches 100% load because of its lack of cache and selection. The net effect of caching is measured by the second configuration. The hit ratio varies from 51.8% to 55.9%, according to the cache size. This is a very good result, as an infinite cache would reach a 57% hit ratio on our query-log, with more than 84,000 unique entries. Interestingly, only about 54,000 of this cache entries are read after their creation, because about 30,000 queries are one-timer. Some recent studies are trying to address this issue. In any case, the cache reduces the load of about 40% (from 100% down to 58.1%).

The next strategies are very strong in reducing the load, at the cost of a reduction in result quality: by performing collection selection, we can reduce the number of queries that reaches each server, but clearly some results will be lost. First, we tested a fixed collection selection, that polls 1, 2, 4 and 8 servers.[2] We measured that, in these configurations, the peak load is about 21.1%, 32.5%, 43.9% and 55.5% respectively. These values were used as threshold for the configurations with dynamic selection (with and without incremental cache).

Even if the fixed collection selection brings some benefit, with a very nice trade-off of coverage vs. load, the most impressive results are reached with load-driven selection. With a very limited load (21.1%), the load-driven collection selection strategy can go over 57% coverage. The incremental cache is able to give another boost, by improving the coverage of frequent queries: the average coverage raises to 65% - 67% (varying with the cache size).

Figure 2 shows the effect of the advanced strategies on coverage, with a cache of 32000 entries. When using the fixed collection selection strategy, the shown load levels are reached by polling 1, 2, 4 and 8 servers. In these configurations, the coverage ranges from 37% to 75%. With the dynamic collection selection, coverage improves clearly, ranging from 58.8% to 83.5%. With the incremental cache, repeated queries retrieve more results, from more servers, improving the perceived quality of results: the more common a query is, the more servers will be used to answer to it. The coverage improves radically, ranging from 67.6% to 87.1%.

Clearly, the biggest benefits are measured with small load threshold levels. When the threshold level is higher, we can retrieve more results with each approach, reducing the differences. With very small load threshold, 21%, we can still retrieve more than 2/3 of the results we would get from the full index. In this configuration, each server is holding a very

[2] Clearly, by selecting all 17 servers, we are back to the previous configuration, with no selection. By selecting 16 servers, still there is no collection selection, but we skip all the results coming from the overflow cluster.

Cache size	Coll. selection	Incremental	Coverage %	Hit-Ratio %	Max Load %
0	None		100	0	100.0
8000	None		100	51.8	58.3
8000	Fixed (1)		37	51.8	21.1
8000	Fixed (2)		47	51.8	32.3
8000	Fixed (4)		59	51.8	43.0
8000	Fixed (8)		75	51.8	54.6
8000	Fixed (17)		100	51.8	58.3
8000	Load-dr. (21.1)		57	51.8	21.1
8000	Load-dr. (32.5)		65	51.8	32.5
8000	Load-dr. (43.9)		72	51.8	43.9
8000	Load-dr. (55.5)		82	51.8	55.5
8000	Load-dr. (21.1)	Increm.	65	51.8	21.1
8000	Load-dr. (32.5)	Increm.	72	51.8	32.5
8000	Load-dr. (43.9)	Increm.	78	51.8	43.0
8000	Load-dr. (55.5)	Increm.	85	51.8	52.8
16000	None		100	53.7	58.2
16000	Fixed (1)		37	53.7	21.2
16000	Fixed (2)		47	53.7	32.1
16000	Fixed (4)		59	53.7	42.1
16000	Fixed (8)		75	53.7	54.3
16000	Load-dr. (21.1)		57	53.6	21.1
16000	Load-dr. (32.5)		65	53.7	32.5
16000	Load-dr. (43.9)		73	53.7	43.9
16000	Load-dr. (55.5)		82	53.7	55.5
16000	Load-dr. (21.1)	Increm.	65	53.6	21.1
16000	Load-dr. (32.5)	Increm.	72	53.7	32.5
16000	Load-dr. (43.9)	Increm.	78	53.7	43.0
16000	Load-dr. (55.5)	Increm.	86	53.7	52.8
32000	None		100	55.9	58.2
32000	Fixed (1)		37	55.9	21.2
32000	Fixed (2)		47	55.9	31.8
32000	Fixed (4)		59	55.9	42.2
32000	Fixed (8)		75	55.9	54.3
32000	Load-dep (21.1)		58.8	55.9	21.1
32000	Load-dep (32.5)		66.7	55.9	32.5
32000	Load-dep (43.9)		74.2	55.9	43.9
32000	Load-dep (55.5)		83.5	55.9	55.5
32000	Load-dep (21.1)	Increm.	67.6	55.9	21.1
32000	Load-dep (32.5)	Increm.	74.6	55.9	32.5
32000	Load-dep (43.9)	Increm.	80.3	55.9	43.0
32000	Load-dep (55.5)	Increm.	87.1	55.9	52.8

TABLE IV
SUMMARY OF EXPERIMENTAL RESULTS

small part of the collection, about 1/32, and is polled every five queries. Alternatively, with a peak load of 32.5% (less than 1/3), our system covers almost 75% results.

The incremental cache has also a very beneficial effect for queries with hard collection selection. If a query is composed of terms not present in the training set, PCAP will not be able to make a strong choice, and the selection will be ineffective. If the query is popular, subsequent hits will retrieve more results, reducing the effects of a wrong selection.

VIII. Conclusions

In this paper, we analyzed the design space of a distributed IR system featuring caching and collection selection. We described different approaches to collection selection, designed to reduce the computing load on the underlying IR system. Our load-driven collection selection system can poll, with decreasing priority, less promising servers, this way exploiting all the available computing load.

Also, the cache can be designed to interact with the selection

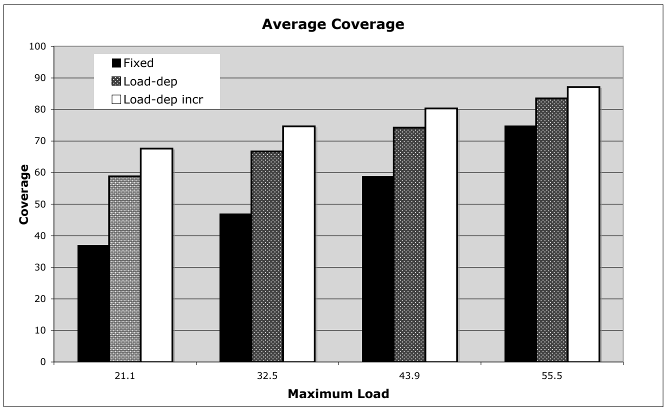

Fig. 2. Coverage comparison, with different strategies, with a LRU cache of 32000 entries. For the fixed strategy, the load levels correspond to polling 1, 2, 3 and 4 servers.

system, by updating results in an incremental way. Every time there is a cache hit, further servers are polled, and their results are added to the cached entries. An incremental cache, over time, stores non degraded results for frequent queries, as all servers will be polled on subsequent hits.

The results are dramatic. We used PCAP to move 50% of our base in an overflow server, and then we partitioned the remaining into 16 clusters: each server is holding about 1/32 of the base. By limiting the peak load to about 20% (i.e. by sending, on average, one query out of five to each sub-server), we are able to cover more than 2/3 of the results we would retrieve from a centralized index, with the full collection. This result is reached by using a small LRU cache (32000 elements), load-driven collection selection and incremental result updating. Alternatively, we can cover 3/4 of the results with a peak load of 32.5%.

We believe that collection selection can be a key element in improving the performance and scalability of modern search engines, by offering a very nice trade-off between result quality and query cost (load). It is also able to adapt dynamically to load peaks, by temporarily reducing the result quality.

This strategy can also be used on bigger systems, on each sub-collection. Commercial systems can index billions of pages, with sub-clusters holding millions at a time. Our strategy can be used on each sub-cluster, in a hierarchical architecture: each set of millions of documents can be partitioned and then collection selection can be used to reduce the computing load.

Acknowledgments: This work was partially supported by XtreemOS, an Integrated Project supported by the European Commission's IST program (#FP6-033576).

References

[1] Claudine Santos Badue, Ramurti Barbosa, Paulo Golgher, Berthier Ribeiro-Neto, and Nivio Ziviani. Distributed processing of conjunctive queries. In *HDIR '05: Proceedings of the First International Workshop on Heterogeneous and Distributed Information Retrieval (HDIR'05)*, SIGIR 2005, Salvador, Bahia, Brazil, 2005.

[2] Ricardo Baeza-Yates, Carlos Castillo, Flavio Junqueira, Vassilis Plachouras, and Fabrizio Silvestri. Challenges in distributed information retrieval (invited paper). In *ICDE*, 2007.

[3] L. A. Barroso, J. Dean, and U. Hölze. Web search for a planet: The google cluster architecture. *IEEE Micro*, 22(2), Mar/Apr 2003.

[4] Steven M. Beitzel, Eric C. Jensen, Abdur Chowdhury, David Grossman, and Ophir Frieder. Hourly analysis of a very large topically categorized web query log. In *SIGIR*, 2004.

[5] Richard Bellman. *Adaptive Control Processes*. Princeton University Press, Princeton, NJ., 1961.

[6] S. Brin and L. Page. The Anatomy of a Large–Scale Hypertextual Web Search Engine. In *Proceedings of the WWW7 conference / Computer Networks*, volume 1–7, pages 107–117, April 1998.

[7] Fidel Cacheda, Vassilis Plachouras, and Iadh Ounis. A case study of distributed information retrieval architectures to index one terabyte of text. *Inf. Process. Manage.*, 41(5):1141–1161, 2005.

[8] J.P. Callan, Z. Lu, and W.B. Croft. Searching Distributed Collections with Inference Networks. In E. A. Fox, P. Ingwersen, and R. Fidel, editors, *Proceedings of the Eighteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 21–28, Seattle, WA, July 1995. ACM Press.

[9] N. Craswell, P. Bailey, and D. Hawking. Server Selection on the World Wide Web. In *Proceedings of the Fifth ACM Conference on Digital Libraries*, pages 37–46, 2000.

[10] I. S. Dhillon, S. Mallela, and D. S. Modha. Information-theoretic co-clustering. In *Proceedings of The Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining(KDD-2003)*, pages 89–98, 2003.

[11] Tiziano Fagni, Raffaele Perego, Fabrizio Silvestri, and Salvatore Orlando. Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Trans. Inf. Syst.*, 24(1):51–78, 2006.

[12] David Hawking and Paul Thistlewaite. Methods for Information Server Selection. *ACM Transactions on Information Systems*, 17(1):40–76, 1999.

[13] Bernard Jansen and Amanda Spink. How are we searching the World Wide Web? A comparison of nine search engine transaction logs. *Inf. Proc. & Management*, 42:248–263, 2006.

[14] Bernard J. Jansen, Amanda Spink, Judy Bateman, and Tefko Saracevic. Real life information retrieval: a study of user queries on the web. *SIGIR Forum*, 32(1):5–17, 1998.

[15] Ronny Lempel and Shlomo Moran. Predictive caching and prefetching of query results in search engines. In *WWW*, 2003.

[16] A. MacFarlane, J. A. McCann, and S. E. Robertson. Parallel search using partitioned inverted files. In *SPIRE '00: Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE'00)*, page 209, Washington, DC, USA, 2000. IEEE Computer Society.

[17] Evangelos P. Markatos. On Caching Search Engine Query Results. *Computer Communications*, 24(2):137–143, 2001.

[18] Alistair Moffat and Justin Zobel. Information Retrieval Systems for Large Document Collections. In *Proceddings of the Text REtrieval Conference*, pages 85–94, 1994.

[19] J. Nielsen. Information foraging: Why google makes people leave your site faster. Available at: http://www.useit.com/alertbox/20030630.html, 2003.

[20] Stefan Podlipnig and Laszlo Boszormenyi. A survey of web cache replacement strategies. *ACM Comput. Surv.*, 35(4):374–398, 2003.

[21] D. Puppin, F. Silvestri, and D. Laforenza. Query-driven document partitioning and collection selection. In *Infoscale 2006*, 2006.

[22] Vijay V. Raghavan and Hayri Sever. On the reuse of past optimal queries. In *Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 344–350, 1995.

[23] S. E. Robertson and S. Walker. Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In *SIGIR '94: Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 232–241, New York, NY, USA, 1994. Springer-Verlag New York, Inc.

[24] Patricia Correia Saraiva, Edleno Silva de Moura, Nivio Ziviani, Wagner Meira, Rodrigo Fonseca, and Berthier Ribeiro-Neto. Rank–Preserving Two–Level Caching for Scalable Search Engines. In *Proceedings of the SIGIR2001 conference*, New Orleans, LA, September 2001. SIGIR, ACM.

[25] L. Si and J. Callan. Relevant document distribution estimation method for resource selection, 2003.

[26] C. Silverstein, M. Henzinger, H. Marais, and M. Moricz. Analysis of a very large web search engine query log. In *ACM SIGIR Forum*, pages 6–12, 1999.

[27] Yinglian Xie and David R. O'Hallaron. Locality in search engine queries and its implications for caching. In *INFOCOM*, 2002.

[28] Xu, Jinxi, and W.B. Croft. Effective Retrieval with Disributed Collections. In *Proceedings of SIGIR98 conference*, Melbourne, Australia, August 1998.

[29] Jinxi Xu and W. Bruce Croft. Cluster-Based Language Models for Distributed Retrieval. In *Research and Development in Information Retrieval*, pages 254–261, 1999.

[30] G. Zervas and S.M. Rger. The curse of dimensionality and document clustering. In *the IEEE Searching for Information: AI and IR Approaches*, 1999.