

# **IXE and R.a.i.s.: a distributed solution for spidering, indexing and searching the Web using a 'Redundant Array of Inexpensive Servers'**

---

Document Version: 1.0  
Contact: [ixe@ideare.com](mailto:ixe@ideare.com)  
Written by: [spagnolo@ideare.com](mailto:spagnolo@ideare.com)  
Revised by: [savona@ideare.com](mailto:savona@ideare.com)  
Approved by: [gulli@ideare.co](mailto:gulli@ideare.co)

## Indice

|     |                               |    |
|-----|-------------------------------|----|
| 1.  | Introduction .....            | 3  |
| 2.  | Base logical components ..... | 3  |
| 3.  | Architecture .....            | 4  |
| 3.1 | Spidering and Indexing.....   | 6  |
| 3.2 | Merging .....                 | 7  |
| 3.3 | Search .....                  | 9  |
| 3.4 | Multisearch .....             | 11 |
| 4.  | Distributed Searches .....    | 11 |
| 4.1 | Related problems .....        | 12 |
| 4.2 | Partitioning data.....        | 13 |
| 5.  | Multisearch servers.....      | 13 |

# IXE and R.a.i.s.: Solutions for web spidering, indexing and searches using a ‘Redundant Array of Inexpensive Servers’

## 1. Introduction

The information retrieval engine IXE integrated with spidering, indexing and search methods for web will be described in this document. The solutions are of the distributed type and are based on a policy of reaching high levels of performance and stability whilst employing relatively inexpensive hardware on a Linux platform. IXE runs also runs on Windows, Solaris and Alpha. ‘Ad hoc’ solutions for these architectures can be designed by following the principles for Linux contained in this paper.

The description of the architecture and steps required for index presume a basic knowledge of IXE and applications which use it. For further reference please consult the papers<sup>1</sup>.

## 2. Base logical components

The indexing and search systems that we are describing can be divided into four base logical components:

- *Spidering and indexing.* The spidering hosts index information obtained from web pages in network, working in parallel.
- *Merging.* The indexes produced by the spiders undergo merge functions to created indexes of pre-defined sizes. (*merged-index*).
- *Index distribution.* Merged indexes are distributed to search servers where they are included in a *multicollection* and assigned a priority.

---

<sup>1</sup> For further information please consult: “How to write a search application on IXE. Basic Introduction: the video index case”, “How to write a search Application on IXE. Advanced introduction: the web index case”, “Ranking In-Document, Off-Document and Persistent Vectors”.

- *Parallel searches.* Procedures must be made for the distribution of the search load to the servers.

This systems allows for the building and management of a series of indexes which vary over time. At the moment  $t$ , the index  $I_t$  used for the search is in reality a multicollection consisting of  $r$  merged-index:

$$I_t = \{i_t^1, \dots, i_t^r\}$$

The generic merged-index  $i_t^s$  with  $1 \leq s \leq r$  is produced independently of the others. In particular, a generic spider  $s$  produces a series of ‘small’ partial indexes:

$$S\text{-PROD}_t^1, \dots, S\text{-PROD}_t^{n(s)}$$

There is no set number  $n(s)$  of partial indexes that a spider  $S$  can produce. Partial indexes are transferred to a server and combined, using a merge operation, to obtain a merged-index.

$$S\text{-PROD}_t^1, \dots, S\text{-PROD}_t^{n(s)} \rightarrow i_t^s$$

This operation is carried out non-stop, 24 hours-a-day, seven days-a-week. At any given moment,  $t+1$ , the merged index  $i_{t+1}^s$  may be produced by merging the merged-index  $i_t^s$  and partial index or group of partial indexes containing updated information originating from a spider. This preserves the information already contained in the merged-index and new data is added or current data updated. The creation of a merged-index  $i_{t+1}^s$  does not necessarily require all indexes originating from the spider to be updated at the statement  $t+1$ .

We can note that the cardinal  $r$  of the merged-index, forming the multicollection, may change. Furthermore, it is not necessary for all the merged-indexes to be updated simultaneously: a multicollection may contain the merged index  $i_{t+1}^s$  and  $i_t^r$  for the time it takes to build merged-index  $i_{t+1}^r$ .

### 3. Architecture

There are four types of hardware necessary for the indexing system (Fig. 1):

- *Spider machines*, which require the creation of an index for each *gatherer* hosted. Each *gatherer* is assigned a group of sites which it spiders, gathering information about the relative web pages.
- *Merge machines*, which only carry out merge operations on the indexes produced by the spiders in order to produce large indexes (*merged-index*).

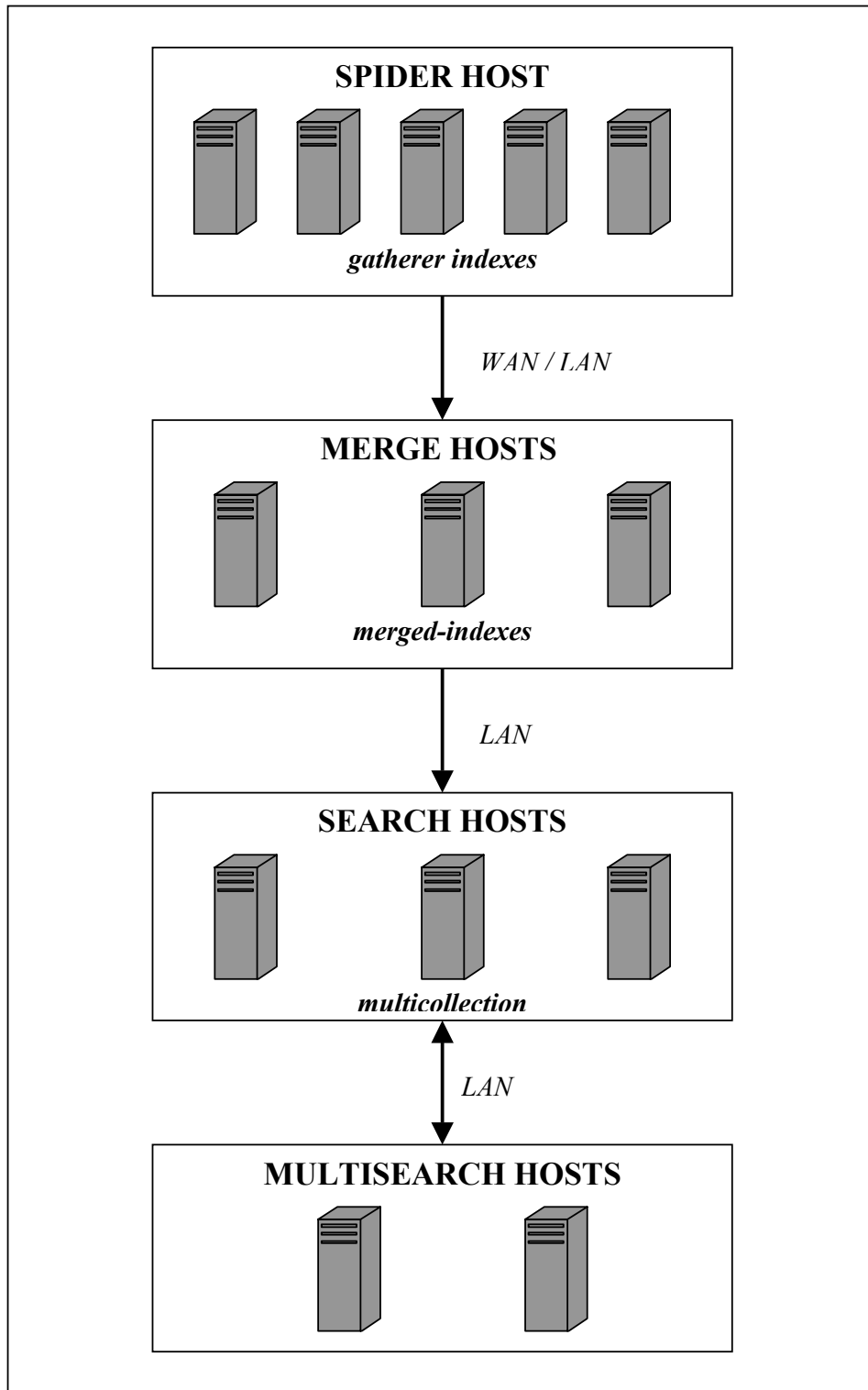


Figure 1

- *Search* machines, which contain the global index (a *multicollection* of merged-indexes) and reply to user queries.
- *Multisearch* machines which channel user queries to the appropriate search servers.

For optimisation some servers may perform more than one operation at the same time. For example some spider machines may perform some merge operations or some merge machines may carry out searches.

### 3.1 Spidering and Indexing

Spider hosts are low cost PCs running Linux. The initial indexing is carried out on these machines. Each spider hosts a number of *gatherers* and for each of these spidering consists of the gathering of web pages starting from a number of prominent web sites. Spidering ends with the creation, for each gatherer, of a *PRODUCTION.gdbm* that contains information taken from the network. The IXE indexing application is capable of indexing information contained in the *PRODUCTION.gdbm*. As soon as the index has been produced it is transferred to a machine set for merge operations. Here are the details of operations carried out by each spider for each gatherer:

- *Creation of the index.* IXE's indexing application is run to create an index starting from *PRODUCTION.gdbm*.
- *Check on index.* The index merge is calculated against itself (*automerge*) thereby checking the index. This test also measures the size of the index produced which for reasons of efficiency is best kept below a limit of 4 Gbyte on Linux.
- *Sending the index.* The spider sends a *tar* archive containing: the index, the MD5 value calculated on the index and list of sites corresponding to the indexed documents to the merge machine. The transfer is carried out using HTTP protocol.

Generally spidering and indexing operations are carried out in parallel by hundreds of host spiders operating independently (Fig. 2).

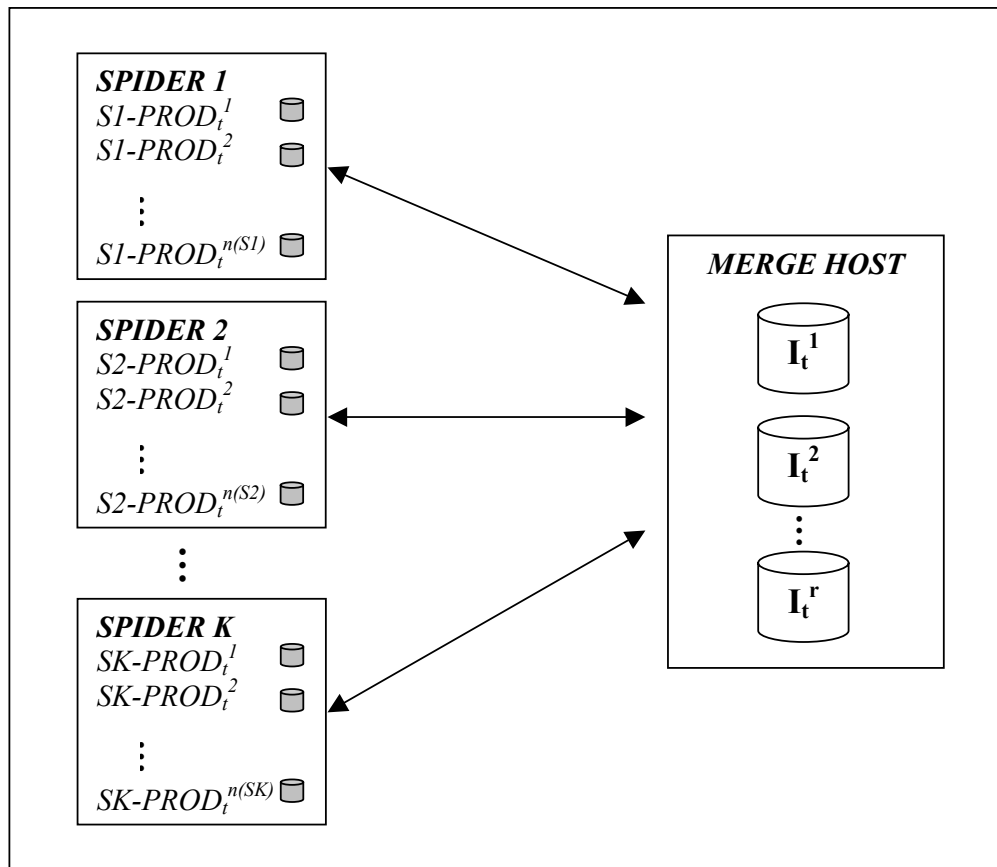


Figure 2. Indexes produced by the host spider are transferred to the server which carries out merge operations.

### 3.2 Merging

Machines used for merging are also low cost and run Linux. The minimum requirements for these machines are:

- Hard disk equal to the space required by the full size of the indexes produces and a predetermined margin.
- Large amount of RAM (1Gbyte)

It is the task of merge machines to gather indexes from spiders and build larger indexes (merged-index) which will form part of the multcollection used by search servers. The ideal size of a merged-index has been set, on machines running Linux, to around 4 Gbyte. For this reason merge operations estimate the size of the merge index to be produced and

ensure that they remain within the 4 Gbyte limit. The size of partial indexes, produced by the spider, may vary through time and consequently so will the merged-indexes produced by the merge machines. To cope with this variable the **relationship** between spider produced indexes and merged-indexes is dynamic and managed by information contained in a database.

In detail the operations performed on merge machines are:

- *Extraction of files from archives.* Merge machines receive data sent from the spider in the form of a tar archive containing indexes, an MD5 code and a list of indexed sites.
- *Selection of the merged-index.* The index coming from the spider must become a ‘candidate’ for inclusion in a merged-index existing on the machine. The merged-index is selected and the index is added to a temporary area where it remains whilst waiting to become a merge object.

At the time of the merge operation the ‘candidate’ indexes may contain new documents or url references already existing in the merged-index. In the first case documents are simply added to the merged-index whilst in the second case they are updated by substituting the old version of the reference.

At regular intervals checks are made on indexes waiting to be updated to the relative merged-index. The necessary operations are (fig. 3):

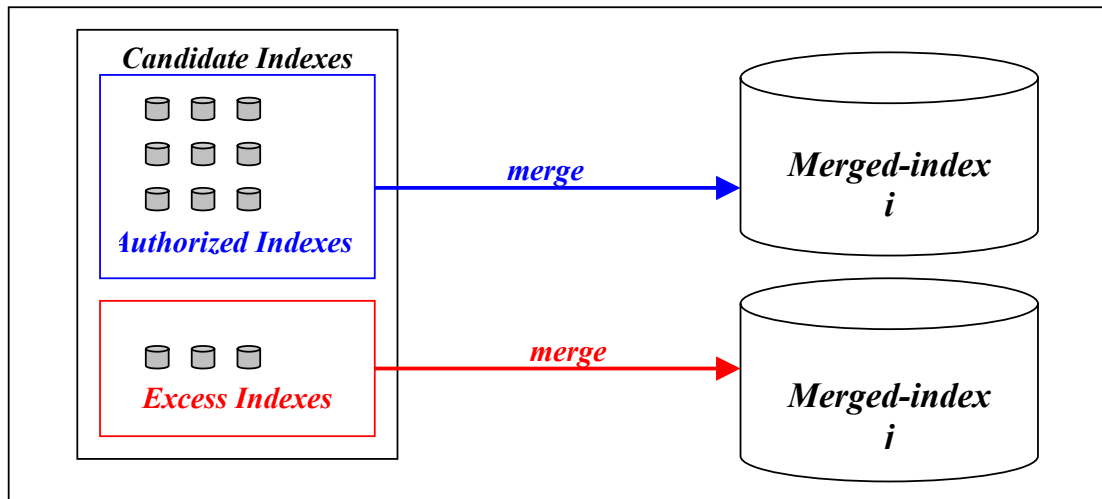


Figure 3. Creation and updating of merged-indexes

1. It is necessary to decide which of the candidate indexes may be updated without exceeding the maximum size allowed for merged-indexes.
2. The merged-indexes and candidate indexes identified in step 1 are merged
3. If an excess number of indexes in the collection is identified in step 1 these are redistributed in other pre-existing merged-indexes or grouped to form a new merged-index.

At set times, merge operations are temporarily suspended to distribute the merged-indexes on the search machines. The distribution of the indexes is done by subdividing the indexes according to country of origin of the documents (tdl) and each group is assigned to one or more search servers.

Merge operations may be carried out in parallel to reduce update times for the total index. This is possible because the servers operate independently.

### 3.3 Search

In keeping with the architecture so far described, the search phases also use low-cost PCs running Linux. This is enough to manage user-formulated queries to the search engine. This system uses multithread posix servers made available by the IXE library.

Merged-indexes are distributed on search servers according to shared tdl dns. In this way is possible to forecast national and generic searches by simply choosing which server to forward the user formulated-query to.

Each search server carries out the search on a multicollecion of merged-indexes  $I_t = \{i_t^1, \dots, i_t^r\}$ . Even if priorities are not used explicitly, IXE implicitly assigns priorities to the merged-index. The results are returned, in order from the merged-index  $1 \dots r$ . In this way, if the number of results obtained from the first merged-indexes is sufficient the remaining ones are not parsed. It is important to note:

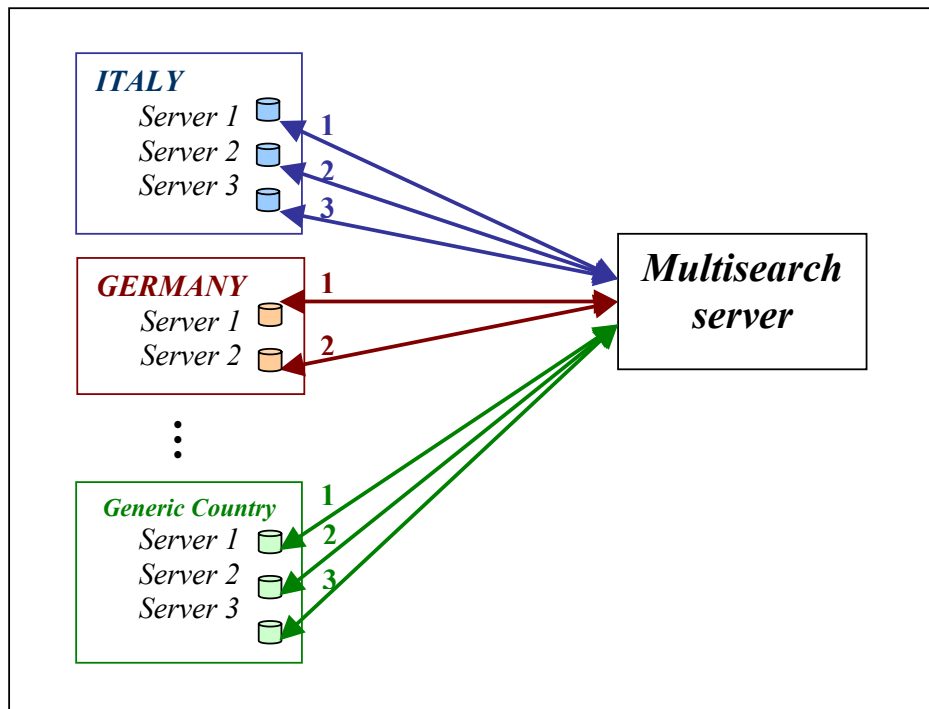
- It is possible to define an index pattern and optimised search for particular objectives that the application is designed for<sup>2</sup>.
- The quality of results can be improved by first inserting indexes containing more relevant documents in into the multicollecion.
- It is possible to locate results less prominent results or those with less recent information lower down in the multicollecion.

---

<sup>2</sup> For further information please consult: “*IXE: examples of search patterns suitable for the creation of custom applications for indexing and searches*”

- It is possible to color individual words taken from a document and assign these a weight or “tag” according to the context in which they are found<sup>3</sup>.
- It is possible to assign an explicit priority to the indexes<sup>4</sup>.

It is possible to use *Off-Document Rank*<sup>5</sup> techniques which allow for the calculation of the document’s rank in the index. This may even be based on factors outside the structure of the document itself. For example, ranking functions may be based on the number of hits the document has had (*user popularity*), or the number of links pointing to the document (*link popularity*) or other criteria which give some indication as to the authoritativeness of the information contained in the index.



**Figure 4. R.a.i.s.:** Each group consists of a number of servers each of which has a copy of the index.

---

<sup>3</sup> For further information please consult: “*How to write a custom Document Reader using IXE*”

<sup>4</sup> For further information please consult: “*How to write a search application on IXE. Advanced introduction: the web index case*”

<sup>5</sup> For further information please consult: “*Ranking In-Document, Off-Document and Persistent Vectors*”

Search operations are performed in parallel, using partial replication of the indexes, to guarantee fault tolerance and by adopting a Round-Robin policy (if necessary this can be weighted) to distribute the load (fig. 4). Essentially the indexes are subdivided into groups, which in this case, correspond to country of origin, but which can be based on other criteria. A number of search servers exist, each of which, contains a copy of the indexes. The multisearch server distributes the search requests to the appropriate group chooses the server for the results in accordance with the weighting established by the Round-Robin policy. This enables IXE to take advantage of R.a.i.s. architecture. The search servers also guarantee a high level of performance thanks to the use of caches for all types of local information (lexis, posting lists, persistent vectors).

### **3.4 Multisearch<sup>6</sup>**

Multisearch machines communicate with the front-end which receives a user-defined query and route these to an appropriate search server.

In the system we are describing, in addition to searches on the world wide web, country-wide searches have been catered for. To this end the merged-indexes on search servers are grouped by country. It is possible to employ different portioning criteria that are more suitable to particular needs.

## **4. Distributed Searches**

The use of a workstation/server network for indexing and distributed searches presupposes that certain choices have been made concerning the portioning of information.

There are essentially two strategies:

*a) Word partitioning:*

The group of indexed documents are subdivided according to words. The corresponding words, posting lists and document store are assigned to different servers.

*b) Document partitioning:*

The group of documents are subdivided and each subgroup is assigned to a server.

In addition, there are a range of solutions that are a compromise between these two options.

---

<sup>6</sup> The IXE multisearch will be released at the end of 2001 with IXE version 1.4. For further information please refer to the document "IXE Roadmap".

## 4.1 Related problems

The architecture we are describing is based on the partitioning of document indexes. The choice was determined by the need to maximise the varying size and stability of the system. The option of word partitioning requires solutions to be found to the following problems:

- *Static balancing of the load*

In the distribution of data it is necessary to have a strategy for the ‘static’ balancing of the load that takes into consideration certain recurring terms have a particularly long posting list (for example, the word ‘sex’ will certainly turn up in a large number of documents).

- *Adaptive strategies for balancing the load*

The frequency of a generic word being a search object may vary through time, on the basis of unpredictable factors, leading to an increase in traffic on one or more nodes. For example ‘anthrax’ will have seen an exponential increase in searches in recently. To resolve these types of problems it is necessary to have an adaptive strategy for balancing the load.

- *Management of memory cache*

Strategies for balancing the load must be compatible with the management of memory cache in search servers.

- *Multikeyword searches*

Problems may exist for searches on more than one word. Solution a) requires the existence of a sophisticated query router capable of dealing with this type of query. It must be able to distribute requests for each word to the appropriate server and calculate the Boolean operator required on the group of documents returned, each of which will certainly contain one of the search words. Where a query consists of an AND for the words ‘red’ and ‘code’ the router must send two requests to two different servers, one request for each word. Once the replies from both servers have been retrieved the router must calculate the documents’ intersection containing the word ‘red’ and those containing the word ‘code’. If this intersection is null the router must request a new group of documents from both the servers concerned. This leads to increased network traffic that could be reduced, though not altogether avoided, by taking specific steps. Similar situations may often occur (e.g. queries using proximity, queries using the NOT operator between search words and queries with prefixes).

- *Updating servers*

Solution a) complicates the updating of servers after the introduction of a new index, even if of reduced size. Indeed it is necessary to update a large portion, or in the worst case, all the search servers.

- **Scalability**

In the case where we wish to **scale** the system it is necessary to organise the redistribution of data involving all the search servers. All this obviously requires the resetting of the system to obtain a balancing of the load.

- *Fault tolerance*

Solution a) undoubtedly has a lower fault tolerance than solution b). A server fault compromises the system's ability to reply to a particular group of queries. This does not happen when employing partitioning by document.

## 4.2 Partitioning data

- The examination of the above problems has led to the choice of a partitioning of indexes by document. The advantages of this scheme are:
- Approach b) provides a greater scalability and this is important if we consider the treatment of large indexes. It also provides functional solutions to the above problems.

There is no need for sophisticated routing strategies for any type of query. Traffic problems are reduced by the use of a multithread posix server which takes advantage of local memory caches. Partitioning by document is a good choice for the balancing of the load and should a R.a.i.s server fail it provides high fault tolerance.

## 5. Multisearch servers

Multisearch servers must guarantee a high level of performance. For this the following solutions have been employed:

1. *Multithread servers with asynchronous I/O on sockets*

Multisearch servers accept front-end requests, namely user-formulated queries. These are distributed to search servers which return the replies obtained using asynchronous I/O. To help deal with a large number of incoming queries multisearch servers allocate and keep alive a pool of threads towards which to send front-end queries. Each thread can communicate with the socket, distribute the queries to the relative server and retrieve the replies obtained (Fig. 5).

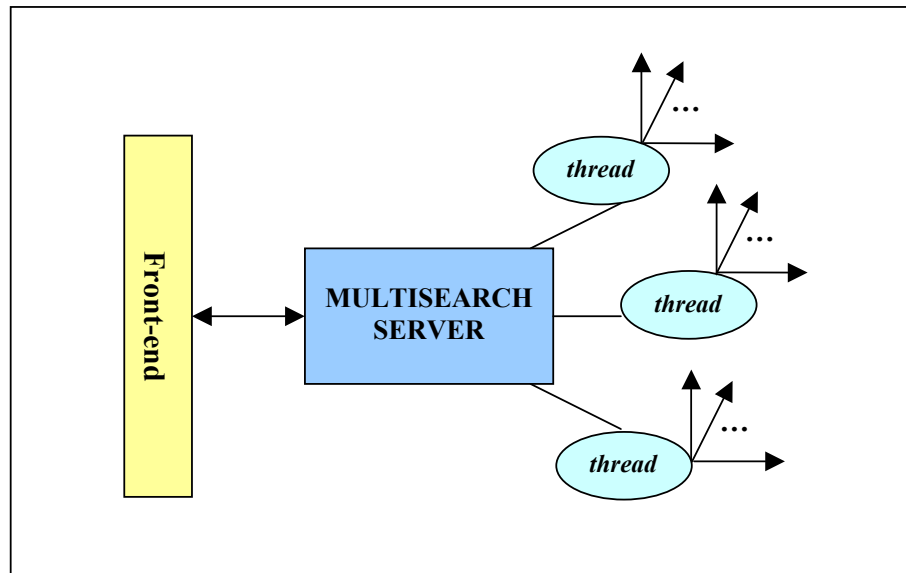


Figure 5. multisearch Server a pool of threads

## 2. Memory cache management policies

To improve performance the memory cache has been fully taken advantage of. The strategies employed being:

- *Hierarchy of indexes held in the cache*

Search and multisearch servers are equipped with a memory cache that can be employed to improve system performance.

- *Lessical cache*

It is possible to use the memory cache to retrieve the following information rapidly:

- *High frequency search words*

Replies to queries of this type can be stored in the memory cache. In this way it is not necessary to forward queries to search servers before returning a reply to the user for the most common queries.

- *Medium frequency search words*

In the case of medium frequency words it is possible to use the cache to keep track of which groups of indexes have, in the past, returned results for the keyword in question (Fig.6). In this way where front-end queries are aimed at all search servers the multisearch server can distribute the request only towards the appropriate search server (*multicast*) rather than towards all servers (*broadcast*).

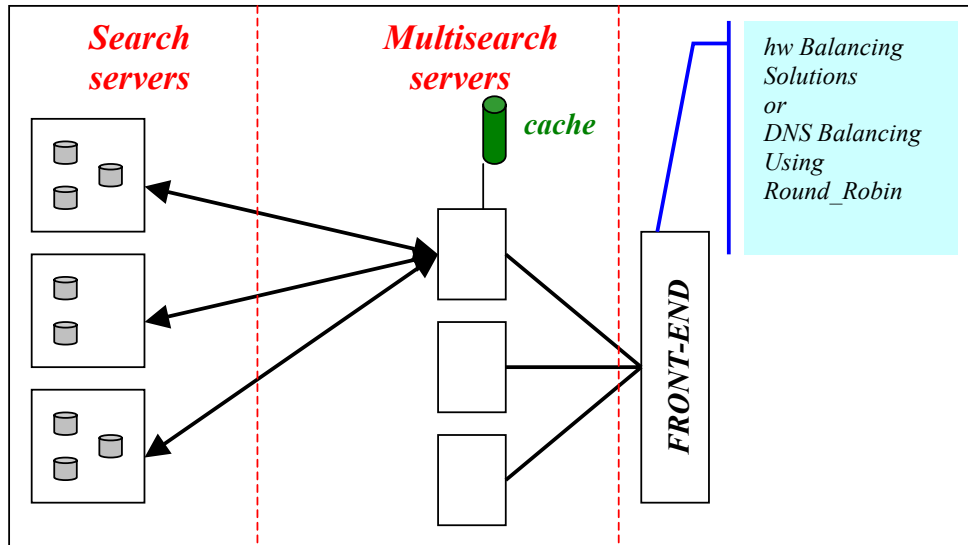


Figure 6. Balancing the load