# An Evaluation of Fault-Tolerant Query Processing for Web Search Engines

Carlos Gomez-Pantoja[1,2], Mauricio Marin[1,3], Veronica Gil-Costa[1,4], and Carolina Bonacic[1]

[1] Yahoo! Research Latin America, Santiago, Chile
[2] DCC, University of Chile, Chile
[3] DIINF, University of Santiago of Chile, Chile
[4] CONICET, University of San Luis, Argentina

**Abstract.** A number of strategies to perform parallel query processing in large scale Web search engines have been proposed in recent years. Their design assume that computers never fail. However, in actual data centers supporting Web search engines, individual cluster processors can enter or leave service dynamically due to transient and/or permanent faults. This paper studies the suitability of efficient query processing strategies under a standard setting where processor replication is used to improve query throughput and support fault-tolerance.

## 1 Introduction

Search engines index very large samples of the Web in order to quickly answer user queries. The data structure used for this purpose is the so-called inverted file [2]. In order to support parallel query processing the inverted file is evenly distributed among $P$ processors forming a cluster of computers. Here either a *local* or *global* approach to indexing can be employed. A number of parallel query processing strategies can be applied to process queries under the local and global indexing approaches. They are based on what we call below *distributed* and *centralized* ranking. Each one has its own advantages under an idealized setting in which processors never fail.

The research question in this paper is what combination of indexing and ranking is the most suitable one under a real-life setting in which processors can leave and re-enter service at unpredictable time instants. We focus on the case in which full quality of service is required, namely the final outcome is not affected by processor failures. More specifically, responding approximated answers to user queries by ignoring the contribution of temporarily out-of-service processors is not an option. This implies that queries (or part of them) hit by failures must be re-executed trying to keep as low as possible their response times.

Note that the complete processing of a query goes through several major steps where usually each step is executed in a different set of cluster processors. The system is dimensioned to make each query last for a very small fraction of a second in each cluster and thereby query re-execution from scratch upon a failure

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 393–404, 2011.
© Springer-Verlag Berlin Heidelberg 2011

is feasible. At the same time, query response time per step cannot be too high since the cumulative sum of response times must not overcome a given upper bound for the latency experienced by users. In this paper we focus on the most costly step for a query, that is, the determination of the top-$R$ documents IDs that best fit the query. In this scenario, our aim is to know how the response time of the individual queries that were active at the instant of a failure is affected and how a failure affects the stability of the whole system in aspects such as overall load balance and query throughput.

To compare the approaches under exactly the same conditions we developed discrete-event simulators validated against actual implementations, which model the essentials of the query processing tasks and uses traces from actual executions to generate the work-load associated with each ranking and indexing strategy.

The remaining of this paper is organized as follows. Section 2 describes the indexing and ranking strategies. Section 3 describes the experimental testbed used to obtain the results presented in Section 4. Conclusions are in Section 5.

## 2 Indexing and Ranking

User queries are received by a broker machine that distributes them among processors (nodes) forming a cluster of $P$ machines. The processors/nodes work cooperatively to produce query answers and pass the results back to the broker. Support for fault-tolerance is made by introducing replication. To this end, and assuming a $P$-processor search cluster, $D-1$ copies for each of the $P$ processors are introduced. The system can be seen as a $P \times D$ matrix in which each row maintains an identical copy of the respective partition (column) of the data.

Local and global indexing: Each column keeps a $1/P$ fraction of an index data structure, called inverted file [2], which is used to speed up the processing of queries. In order to support parallel query processing the inverted file is evenly distributed among the $P$ columns and replicated $D-1$ times along each column. An inverted file is composed of a vocabulary table and a set of posting lists. The vocabulary table contains the set of distinct relevant terms found in the document collection. Each of these terms is associated with a posting list that contains the document identifiers where the term appears in the collection along with additional data used for ranking purposes.

The two dominant approaches to distributing an inverted file [1,9,10,11,15,17,7] among $P$ processors are: (**a**) the document partitioning strategy (also called *local indexing*), in which the documents are evenly distributed among the processors and an inverted index is constructed in each processor using the respective subset of documents, and (**b**) the term partitioning strategy (called *global indexing*), in which a single inverted file is built from the whole text collection to then evenly distribute the terms and their respective posting lists among the processors.

Distributed and centralized ranking: Trying to bring into the same comparison context the strategies reported in the literature is quite involved since almost each research group uses different methods for document ranking, query

processing upon the term, and document partitioned inverted files [1,9,10,15,17]. Some use either intersection (AND) or union (OR) queries [6,8,18]. Others perform exhaustive traversal of posting lists while others do list pruning which can be made on posting lists sorted by term frequency or document ID.

In this work, the ranking of documents is performed using the list pruning method described in [16]. This method generates a workload onto processors that is representative of other alternative methods [2,4,5]. In this case, posting lists are kept sorted by in-document term frequency. The method works by pushing forward a barrier $S_{\max}$ [16]. A given posting list pair (*id_doc*, *freq*) is skipped if the term frequency *freq* in the document *id_doc* is not large enough to overcome the barrier $S_{\max}$. For either local or global indexing, one can apply this method for either distributed or centralized ranking.

For local indexing under distributed ranking, the broker sends a given query to a processor selected in a circular manner. We say this processor becomes the *merger* for this query. A merger receives a query from the broker and broadcasts it to all processors. Then all processors execute the document ranking method and update their barrier $S_{\max}$ using the local posting lists associated with the query terms. After these calculations, each processor sends to the merger its local top-$R$ results to let the merger calculate the global top-$R$ documents and send them to the broker machine. These documents are then sent to a second cluster of computers, which contain the actual text of documents, in order to produce the final answer web-page presented to the user.

For global indexing under distributed ranking, the broker sends the query to one of the processors that contain the query terms (the least loaded processor is selected). This processor becomes the merger for the query. In the case that the query terms are not located all in the same processor, the merger processor sends the query to all other processors containing query terms. Then those processors compute their estimation of the top-$R$ results. In this case, however, the processors that do not contain co-resident terms can only compute an approximation of the ranking because they do not have information about the remote posting lists to increase their respective barriers $S_{\max}$.

In [1], it is suggested that processors should send to the merger at least $6\,R$ results for global indexing. The merger must re-compute the document scores by using information of all terms and then produce the global top-$R$ results. Note that with probability $1/P$ two query terms are co-resident. This probability can be increased by re-distributing terms among processors so that terms appearing very frequently in the same queries tend to be stored in the same processors. This can be made by obtaining term correlation information from actual queries submitted in the recent past. We call this approach *clustered* global indexing.

Centralized ranking [3,12] is different in the sense that now pieces of posting lists are sent to processors to be globally ranked rather than performing local ranking on them in their respective processors. The rationale is that the granularity of ranking is much larger than the cost of communication in current cluster technology and that by performing global ranking the barrier $S_{\max}$ goes up much faster which significantly reduces the overall fraction of posting lists

scanned during ranking. Centralized ranking stops computations earlier than distributed ranking. In practice, this means that each query requires less processor time to be solved.

For local indexing under centralized ranking the broker sends the query to one of the processors selected circularly. This processor becomes the ranker for the query. Then the ranker broadcasts the query to all processors and they reply with a piece of posting list of size $K/P$ for each query term (with $K$ fairly larger than $R$, say $K = 2\,R$).

The ranker merges these pieces of posting lists and execute the ranking method sequentially. After this, for all terms in which the pruning method consumed all the pairs (*id_doc*, *freq*), a new block of $K/P$ pairs per term is requested to the $P$ processors (ranker included) and the same is repeated. The ranking ends up after completing a certain number of these iterations of size $K$ per term.

For global indexing under centralized ranking the broker sends the query to one of the processors circularly selected. This processor becomes the ranker for the query and it might not contain any of the query terms. The reason for this is to favor the load balance of the ranking process (at the expense of communication) which is the most costly part of the solution of a query. Upon reception of a query, the ranker processor sends a request for the first $K$-sized piece of posting list for each query term to each processor holding them. The arriving pairs (*id_doc*, *freq*) are merged and passed through the ranking method sequentially as in the local indexing and centralized ranking case.

The disadvantage of the global indexing approach is the above mentioned AND queries in which a large amount of communication is triggered each time it is necessary to calculate the intersection of two posting lists not located in the same processor. Another difficulty in the global index is the huge extra communication required to construct the index and distribute it among the processors. A practical solution to this problem has been proposed in [11].

Note that in the local indexing approach it is possible to reduce the average number of processors involved in the solution of queries as proposed in [13] by using a location cache. This comes from the fact that one can re-order documents stored in the processors in such a way that they form clusters to which queries can be directed. At search time, when there is not enough information to determine which of the clusters can significantly contribute to the global top-$R$ results, the query is sent to all processors as in the conventional approach. In this paper we study this approach as well. We call it *clustered* local indexing.

Overall, in this paper we investigate how efficient are the different query processing strategies listed in Table 1 under fails of randomly selected processors.

## 3 Experimental Framework

The processing of a query can be basically decomposed into a sequence of very well defined operations [12]. For instance, for local indexing and centralized ranking, the work-load generated by the processing of an one-term query requiring $r$ iterations to be completed and arriving to processor $p_i$ can be represented by the sequence,

**Table 1.** The different strategies for parallel query processing investigated in this paper and the code names used to label them in the figures of Section 4

| *Code* | *Query Processing Strategy* |
|---|---|
| **GIDR** | Global Indexing with Distributed Ranking |
| **CGIDR** | Clustered Global Indexing with Distributed Ranking |
| **LIDR** | Local Indexing with Distributed Ranking |
| **CLIDR** | Clustered Local Indexing with Distributed Ranking |
| **GICR** | Global Indexing with Centralized Ranking |
| **CGICR** | Clustered Global Indexing with Centralized Ranking |
| **LICR** | Local Indexing with Centralized Ranking |
| **CLICR** | Clustered Local Indexing with Centralized Ranking |

**Table 2.** Boldface letters in the expressions stand for the primitive operations Broadcast (B), Fetch (F), Rank (R), Send (S) and Merge (E). Also $M \leq P$ and we use "+" to indicate one or more repetitions of the same sequence of operations.

| *Code* | *Most-likely Sequence of Primitive Operations for Two-Terms Queries* |
|---|---|
| GIDR | $[\,\mathbf{F}(K) \,\|\|\, 2 \rightarrow \mathbf{R}(K) \,\|\|\, 2\,]^{+} \rightarrow \mathbf{S}(6R) \rightarrow \mathbf{E}(12R)^{\langle p_i \rangle}$ (just one term in $p_i$) |
| CGIDR | $[\,2\,\mathbf{F}(K) \rightarrow \mathbf{R}(2K)\,]^{+}$ (both terms are in $p_i$ with high probability) |
| LIDR | $\mathbf{B}(2t)_{P}^{\langle p_i \rangle} \rightarrow [\,2\,\mathbf{F}(K/P) \,\|\|\, P \rightarrow \mathbf{R}(2K/P) \,\|\|\, P\,]^{+} \rightarrow \mathbf{S}(R) \,\|\|\, P \rightarrow \mathbf{E}(PR)^{\langle p_i \rangle}$ |
| CLIDR | $\mathbf{B}(2t)_{M}^{\langle p_i \rangle} \rightarrow [\,2\,\mathbf{F}(K/M) \,\|\|\, M \rightarrow \mathbf{R}(2K/M) \,\|\|\, M]^{+} \rightarrow \mathbf{S}(R) \,\|\|\, M \rightarrow \mathbf{E}(MR)^{\langle p_i \rangle}$ |
| GICR | $[\,\mathbf{F}(K) \,\|\|\, 2 \rightarrow \mathbf{S}(K) \rightarrow \mathbf{R}(2K)^{\langle p_i \rangle}\,]^{+}$ (one term in $p_i$, probability $1-1/P$) |
| CGICR | $[\,2\,\mathbf{F}(K) \rightarrow \mathbf{R}(2K)\,]^{+}$ (both terms are in $p_i$ with high probability) |
| LICR | $\mathbf{B}(2t)_{P}^{\langle p_i \rangle} \rightarrow [\,2\,\mathbf{F}(K/P) \,\|\|\, P \rightarrow 2\,\mathbf{S}(K/P) \,\|\|\, P \rightarrow \mathbf{R}(2K)^{\langle p_i \rangle}\,]^{+}$ |
| CLICR | $\mathbf{B}(2t)_{M}^{\langle p_i \rangle} \rightarrow [\,2\,\mathbf{F}(K/M) \,\|\|\, M \rightarrow 2\,\mathbf{S}(K/M) \,\|\|\, M \rightarrow \mathbf{R}(2K)^{\langle p_i \rangle}\,]^{+}$ |

$$\text{Broadcast}(t)^{\langle p_i \rangle} \rightarrow \Big[\,\text{Fetch}(K/P) \,\|\, P \rightarrow \text{Send}(K/P) \,\|\, P \rightarrow \text{Rank}(K)^{\langle p_i \rangle}\,\Big]^{r}$$

The broadcast operation represents the part in which the ranker processor $p_i$ sends the term $t$ to all processors. Then, in parallel ($\|$), all $P$ processors perform the fetching, possibly from secondary memory, of the $K/P$ sized pieces of posting lists. Also in parallel they send those pieces to the ranker $p_i$. The ranker merges them and performs a ranking of size $K$ repeated until completing the $r$ iterations. Similar workload representations based on the same primitives Broadcast, Fetch, Send, and Rank can be formulated for the combinations of Table 1. The exact order depends on the particular indexing and ranking methods, and basically all of them execute the same primitive operations but with **(i)** different distribution among the processors, **(ii)** different timing as their use of resources is directly proportional to their input parameter (e.g., Send($K$) or Send($R$)$\|\ P$), and **(iii)** different number of repetitions $r$. For the sake of a fair comparison, we perform distributed ranking partitioned in quanta of size $K$ for global indexing and $K/P$ for local indexing (this does not imply an extra cost, it just fixes the way threads compete and use the hardware resources in the simulator). We ignore the cost of merging because it is comparatively too small with

respect to ranking. In Table 2 we show the sequences of primitive operations for the strategies described in Table 1.

For each query we obtained the exact trace or sequence of primitive calls (including repetitions) from executions using query logs on actual implementations of the different combinations for ranking and indexing. Those programs also allow the determination of the relative cost of each primitive with respect to each other. We included this relative cost into the simulator. Apart from allowing a comparison under exactly the same scenario, the simulator with its primitive operations competing for resources, has the advantage of providing an implementation independent view of the problem since results are not influenced by interferences such as programming decisions and hardware among others.

### 3.1 Process-Oriented Discrete-Event Simulator

The query traces are executed in a process-oriented discrete-event simulator which keeps a set of $P \times D$ concurrent objects of class Processor. The simulator keeps at any simulation time instant a set of $q \cdot P$ active queries, each at a different stage of execution as its respective trace dictates. An additional concurrent object is used to declare out of service a processor selected uniformly at random, and another object is used to re-start those processors. These events take place at random time intervals with negative exponential distribution.

For each active query there is one fetcher/ranker thread per processor and one ranker/merger thread for centralized/distributed ranking. To cause cost, the simulator executes query quanta for both centralized and distributed ranking (Table 2). The type of indexing determines the size of the quanta, either $K$ or $K/P$, to be used by each primitive operation and also the level of parallelism, either $P$, number of non-resident terms or none. The quanta can be $K/M$, with $M \leq P$, when clustered local indexing is used. Here the level of parallelism is $M$, where $M$ is the number of processors where the query is sent based on a prediction of how "good" are document clusters stored in those processors for the query (for each query, $M$ is determined from actual execution traces).

The asynchronous threads are simulated by concurrent objects of class Thread attached to each object of class Processor. Competition for resources among the simulated threads is modeled with $P$ concurrent queuing objects of class CPU and *ram-cached* Disk which are attached to each Processor object respectively, and one concurrent object of class Network that simulates communication among processors. Each time a thread must execute a Rank($x$) operation it sends a request for a quanta of size $x$ to the CPU object. This object serves requirements in a round-robin manner. In the same way a Fetch($x$) request is served by the respective Disk object in a FCFS fashion. These objects, CPU and Disk, execute the SIMULA like `Hold(time_interval)` operation to simulate the period of time in which those resources are servicing the requests, and the requesting Thread objects "sleep" until their requests are served.

The concurrent object Network simulates an all-to-all communication topology among processors. For our simulation study the particular topology is not really relevant as long as all strategies are compared under the same set of

resources CPU, Disk and Network, and their behavior is not influenced by the indexing and ranking strategy. The Network object contains a queuing communication channel between all pairs of processors. The average rate of channel data transfer between any two processors is determined empirically by using benchmarks on the hardware described in Section 4.

Also the average service rate per unit of quanta in ranking and fetching is determined empirically using the same hardware, which provides a precise estimation of the relative speed among the different resources. We further refined this by calibrated the simulator to achieve a similar query throughput to the one achieved by the actual hardware. This allows us to evaluate the consequences of a processor failure under a fairly realistic setting with respect to its effects in the throughput and average response time of queries.

The simulator has been implemented in `C++` and the concurrent objects are simulated using `LibCppSim` library [14]. To obtain the exact sequence of primitive operations (Table 2) performed by each strategy (Table 1), we executed `MPI-C++` implementations of each strategy on a cluster of computers using $P=32$ processors and an actual query log. We indexed in various forms a 1.5TB sample of the UK Web and queries were taken from an one year log containing queries submitted by actual users to www.yahoo.co.uk.

### 3.2 Simulating Failures

During the simulation of *centralized* ranking and just before a failure, all processors are performing their roles of rankers and fetchers for different queries. Upon failure of processor $p_i$, all queries for which $p_i$ was acting as ranker must be re-executed from scratch. This is mainly so because the current (potentially large) set of candidate documents to be part of the global top-$R$ results is irreversibly lost for each query in which $p_i$ was their ranker. In this case a new ranker processor for those queries must be selected from one of the $D-1$ running replicas of processor $p_i$ and re-execution is started off. Nevertheless, all pieces of posting lists already sent by the fetchers are at this point cached in their respective processors so the disk accesses are avoided and the extra cost comes only from the additional communication. For other queries, the processor $p_i$ was acting as a fetcher (and sender) of pieces of posting lists. The respective rankers detect inactivity of $p_i$ and send their following requests for pieces of posting lists to one of the $D-1$ running replicas. In both cases (i.e., $p_i$ ranker and fetcher) the replicas are selected uniformly at random.

For *distributed* ranking and just before a failure, the processors are performing merging of local top-$R$ results to produce the global top-$R$ ones for a subset of the active queries. They are also performing local ranking to provide their local top-$R$ results to the mergers of other subset of queries. Upon failure of processor $p_i$, a new replica $p_i$ is selected uniformly at random for each query for which $p_i$ is a merger. The other processors performing local-ranking for those queries send their local top-$R$ results to those replicas of $p_i$ and the replicas selected as mergers must re-execute the local ranking for those queries to finally obtain the global top-$R$ results for the affected queries. The processor $p_i$ was also acting as

local ranker for mergers located in other processors. In that case, the randomly selected replicas $p_i$ recalculate the local top-$R$ for the affected queries and send them to their mergers.

### 3.3 Simulator Validation

Proper tuning of the simulator cost parameters is critical to the comparative evaluation of the strategies. We set the values of these parameters from actual implementation runs of the query processing strategies. These implementations do not support fault tolerance and thereby are not able to exactly reproduce the same conditions before, during and after processor failures. Such an objective would be impossible in practice given the hardware and system software available for experimentation. Even so, this would require us to modify the MPI library protocols to make it able to detect failures without aborting the program in all processors and to selectively re-start queries hit by failures. These protocols have to be application specific as it is a firm requirement that response time of queries must be below a fraction of a second. This excludes MPI realizations prone to scientific and grid computing applications that provide support for processor failures. This also excludes systems such as map-reduce (e.g., Hadoop) which are also intended to perform off-line processing in terms of what is understood as on-line processing in Web search engines.

Nevertheless, the simulator allows us to achieve the objectives of evaluating different failure scenarios. These scenarios start from a situation in which the simulator mimics, in a very precise manner, the steady state regime of the actual program running the respective query processing strategy. From that point onwards processor failures are injected, there is an increase in load for surviving processors, the underlying network must cope with lost connections, and signal re-execution of queries. The cost parameters of the concurrent objects do not change in this case and these objects properly simulate saturation when some sections of the network and processors are overloaded.

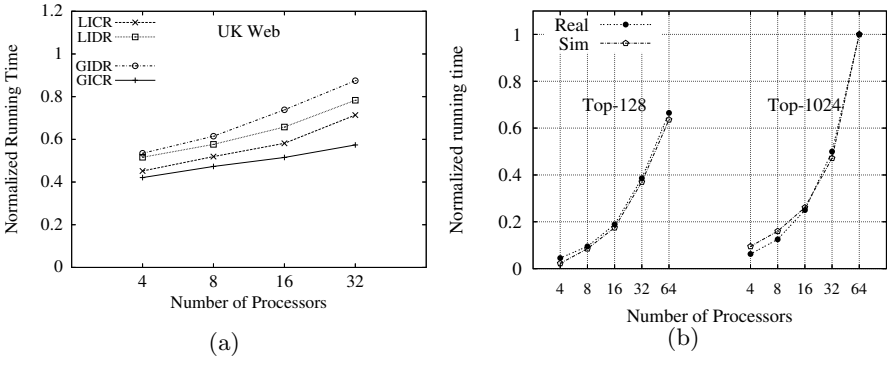


**Fig. 1.** Validation of simulations against actual implementations

Figure 1 shows results for overall running time achieved with actual implementations of the query processing strategies. We injected the same total number of queries in each processor. Thereby, running times are expected to grow with the number of processors. The results from the simulator (omitted in Figure 1.a, differences below 1%) overlap each curve very closely. Figure 1.b shows results both for real execution time and simulation time for the LIDR strategy. These results are for an implementation of the LIDR strategy independent to ours. They were obtained with the Zettair Search Engine (www.seg.rmit.edu.au/zettair) and we adjusted our simulator parameters to simulate the performance of Zettair. We installed Zettair in the 32 processors of our cluster and broadcast queries to all processors using TCP/IP sockets to measure query throughput. Again the simulator is capable of almost overlapping each point in the curves of Figure 1.b. The two sets of curves show results for $K$= 128 and 1024 respectively.

## 4 Comparative Evaluation

In the following, we show results normalized to 1 in order to better understand and illustrate a comparative analysis in terms of percentage differences among the strategies. We assume that main memory of processors is large enough to make negligible disk accesses as it indeed occurs in current search engines.

Figures 2.a, 2.c, 2.e and 2.g show results for centralized ranking (CR) upon global and local indexing (GI and LI respectively) for $P = 32$ and $D = 4$. Figure 2.a shows the average number of queries that are affected by failures for different rates of failures. The $x$-axis values indicate a range of rates of failures from to very high to very low; the value 10 indicates a very high rate whereas 80 indicates a very low rate. Figure 2.c shows the effect in the response time of queries that must be re-executed from scratch upon a failure. Figure 2.e shows the effect in all of the queries processed during the experiments. Figure 2.e shows overall query throughput.

Firstly, these results show that overall queries are not significantly affected by failures in the $P \times D$ arrangement of processors. Secondly, these results indicate that the strategies that aim at reducing the number of processors involved in the solution of individual queries are more effective. This is counter intuitive since as they use less processors, the amount of calculations that must be re-executed is larger than the strategies that use more processors to solve individual queries. However, these strategies are fast enough to let re-executions be performed with a more quiet effect in the surrounding queries not hit by the failure.

The same trend is observed in the strategies that use distributed ranking (DR) upon the global and local indexes (GI and LI). This can be seen in the Figures 2.b, 2.d, 2.f and 2.h. The performance of these strategies is quite behind the centralized ranking strategies. On average they performed at least 5 more iterations during query processing making that the impact of failures were more significant in terms of query throughput degradation. This is reflected in Figures 3.a and 3.b which show the average response time for all queries (3.a) and for queries involved in a failure only (3.b).

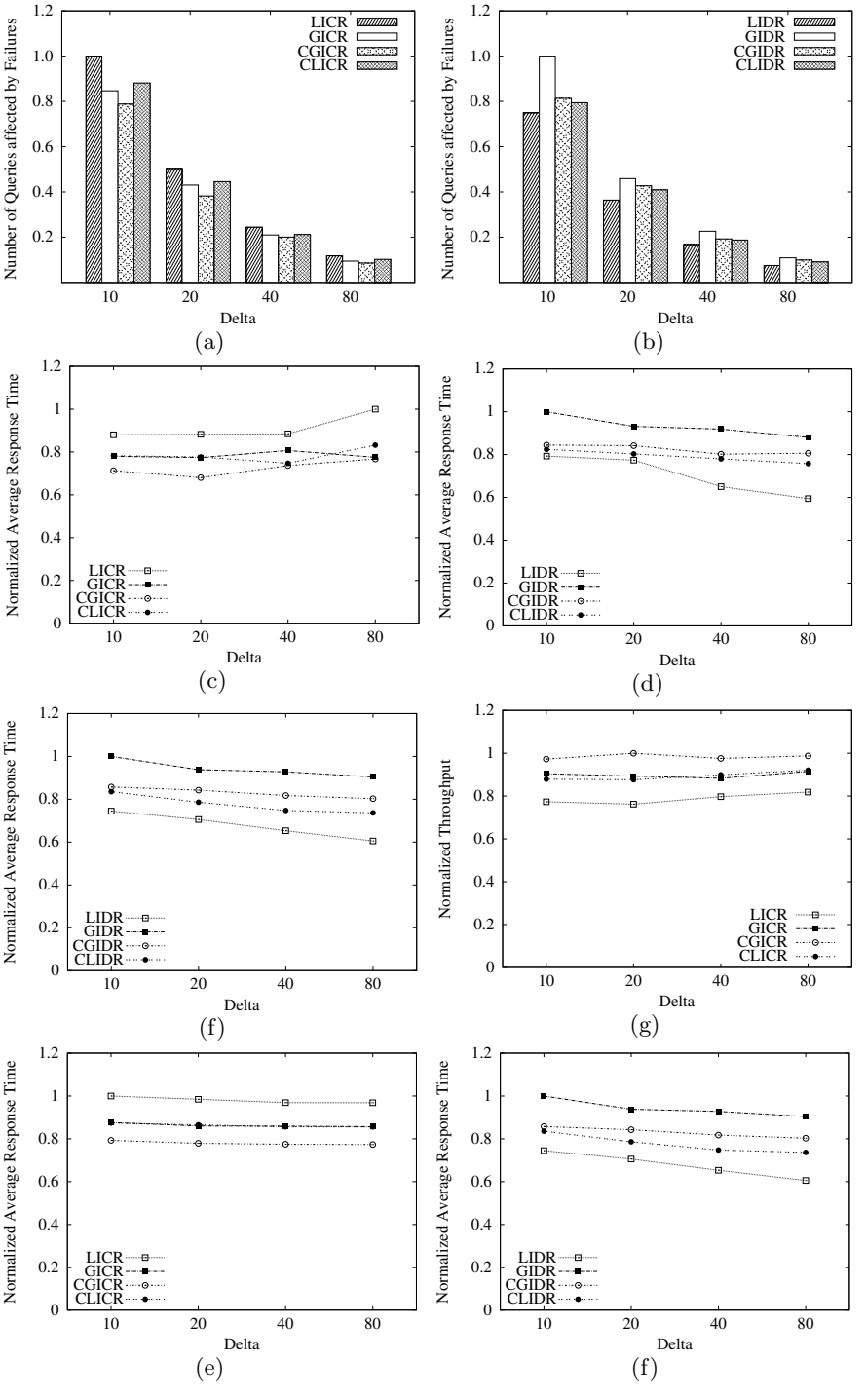


**Fig. 2.** Experiments for centralized and distributed ranking

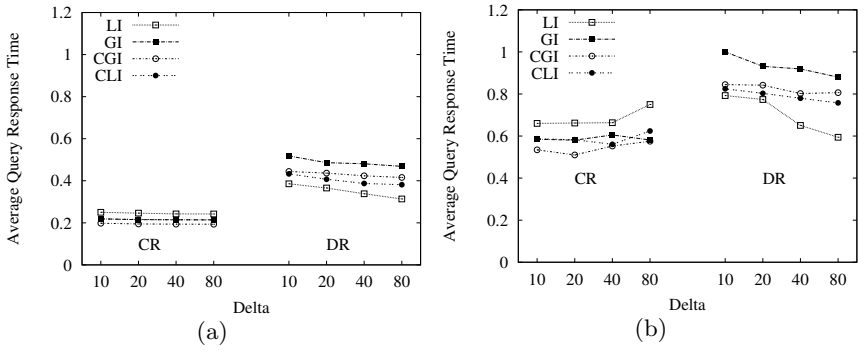


**Fig. 3.** Normalized average response times for all queries (a) and for queries involved in failures (b). Both figures are in the same normalized scale.

## 5 Concluding Remarks

In this paper we have evaluated different strategies for indexing and ranking in Web search engines under a fault-tolerant scenario. We resorted to simulators to re-produce exactly the same conditions for each tested strategy. The simulators were properly tuned with actual implementations of the strategies. Simulations and actual executions agreed within a small difference. For each strategy, traces were collected from the execution of the actual implementations and injected in the respective simulators to cause cost in the simulated time and react upon failures by re-executing the respective traces of affected queries. The failure arrival rate was kept independent of the query processing strategy being simulated.

The results from a set of relevant performance metrics, clearly show that centralized ranking strategies behave better than distributed ranking strategies upon failures. This holds for both local and global indexing. At first glance this appears counter intuitive since distributed ranking gets more processors involved in the document ranking process and thereby upon a failure it is only necessary to re-execute $1/P$-th of the computations, with $P$ being the number of involved processors. In contrast, centralized ranking assigns just one processor to the document ranking process and thereby the query must be re-executed from scratch in another processor. The key point is that centralized ranking is much faster than distributed ranking and this makes the difference in a environment prone to failures since the system quickly gets back into steady state.

In addition, the simulations show that global indexing achieves better performance than local indexing for list-pruning ranking methods under processor failures. This kind of methods have become current practice in major vertical search engines since they reduce the amount of hardware devoted to process each single query. Our results show that global indexing in combination with centralized ranking is able to significantly reduce hardware utilization for disjunctive queries. Conjunctive queries are better adapted to local indexing but in both cases, centralized ranking is a better alternative than distributed ranking.

A combination of distributed and centralized ranking is also possible: a first round can be made using the centralized approach to quickly increase the global $S_{\max}$ barrier for the query, and then the global barrier is communicated to the involved processors so that they can use it to perform distributed ranking from this point onwards. This combination, which reduces overall communication, is expected to be useful in cases in which the ranking method is not able to aggressively prune posting list traversal for all query terms. We plan to study this alternative in the near future.

## References


1. Badue, C., Baeza-Yates, R., Ribeiro, B., Ziviani, N.: Distributed query processing using partitioned inverted files. In: SPIRE, pp. 10–20 (November 2001)
2. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. Addison-Wesley, Reading (1999)
3. Bonacic, C., Garcia, C., Marin, M., Prieto, M.E., Tirado, F.: Exploiting Hybrid Parallelism in Web Search Engines. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 414–423. Springer, Heidelberg (2008)
4. Broder, A.Z., Carmel, D., Herscovici, M., Soffer, A., Zien, J.Y.: Efficient query evaluation using a two-level retrieval process. In: CIKM, pp. 426–434 (2003)
5. Broder, A.Z., Ciccolo, P., Fontoura, M., Gabrilovich, E., Josifovski, V., Riedel, L.: Search advertising using web relevance feedback. In: CIKM, pp. 1013–1022 (2008)
6. Chaudhuri, S., Church, K., Christian König, A.: Liying Sui. Heavy-tailed distributions and multi-keyword queries. In: SIGIR, pp. 663–670 (2007)
7. Ding, S., Attenberg, J., Baeza-Yates, R.A., Suel, T.: Batch query processing for Web search engines. In: WSDM, pp. 137–146 (2011)
8. Falchi, F., Gennaro, C., Rabitti, F., Zezula, P.: Mining query logs to optimize index partitioning in parallel web search engines. In: INFOSCALE, p. 43 (2007)
9. Jeong, B.S., Omiecinski, E.: Inverted file partitioning schemes in multiple disk systems. TPDS 16(2), 142–153 (1995)
10. MacFarlane, A.A., McCann, J.A., Robertson, S.E.: Parallel search using partitioned inverted files. In: SPIRE, pp. 209–220 (2000)
11. Marin, M., Gil-Costa, V.: High-performance distributed inverted files. In: CIKM 2007, pp. 935–938 (2007)
12. Marin, M., Gil-Costa, V., Bonacic, C., Baeza-Yates, R.A., Scherson, I.D.: Sync/async parallel search for the efficient design and construction of web search engines. Parallel Computing 36(4), 153–168 (2010)
13. Marin, M., Gil-Costa, V., Gomez-Pantoja, C.: New caching techniques for web search engines. In: HPDC, pp. 215–226 (2010)
14. Marzolla, M.: Libcppsim: a Simula-like, portable process-oriented simulation library in C++. In: ESM, pp. 222–227. SCS (2004)
15. Moffat, A., Webber, W., Zobel, J., Baeza-Yates, R.: A pipelined architecture for distributed text query evaluation. Information Retrieval (August 2007)
16. Persin, M., Zobel, J., Sacks-Davis, R.: Filtered document retrieval with frequency-sorted indexes. JASIS 47(10), 749–764 (1996)
17. Xi, W., Sornil, O., Luo, M., Fox, E.A.: Hybrid partition inverted files: Experimental validation. In: Agosti, M., Thanos, C. (eds.) ECDL 2002. LNCS, vol. 2458, pp. 422–431. Springer, Heidelberg (2002)
18. Zhang, J., Suel, T.: Optimized inverted list assignment in distributed search engine architectures. In: IPDPS (2007)