On-Line Multi-Threaded Processing of Web User-Clicks on Multi-Core Processors

Carolina Bonacic[1], Carlos Garcia[1], Mauricio Marin[2], Manuel Prieto[1], and Francisco Tirado[1]

[1] Depto. Arquitectura de Computadores y Automatica
Universidad Complutense de Madrid
cbonacic@fis.ucm.es, {garsanca, mpmatias, ptirado}@dacya.ucm.es

[2] Yahoo! Research Latin America
Universidad de Santiago de Chile
mmarin@yahoo-inc.com

Abstract. Real time search — a setting in which Web search engines are able to include among their query results documents published on the Web in the very recent past — is a clear evidence that many of the off-line computations performed so far on conventional search engines need to be moved to the on-line arena. This is a demanding case for parallel computing since it is necessary to cope efficiently with thousands of concurrent read and write operations per unit time, all requiring latency times within a fraction of a second. To our knowledge, computations related to capturing user preferences through their clicks on the query result webpages and include this feature in the document ranking process are currently performed in an off-line manner. This is effected by pre-processing very large logs containing millions of queries submitted by actual users in a time scale of days, weeks or even months. The outcome is score data for the set of documents indexed by the search engine which were selected by users in the past. This paper studies the efficiency of this process in the on-line setting by evaluating a set of strategies for concurrent read/write operations executed on a multi-threaded multi-core architecture. The benefit of efficient on-line processing of user clicks is making it feasible to include user preference in document ranking also in a real-time fashion.

1 Introduction

Conventional Web Search Engines track user clicks performed on the URLs listed on the webpages containing search results to improve the quality of the document ranking process. User clicks are monitored along time to detect document popularity trends so that ranking can be updated accordingly to refine the results of subsequent queries; previously high-ranked pages that are not attracting user clicks are eventually demoted, while previously low-ranked pages that capture the interest of visitors are rewarded with a rank boost.

However, most of the optimizations to the ranking process that are based on click rates are still performed in an off-line manner. This means that the effects of previous clicks on the present ranking process only become visible at regular intervals of the order of hours or even days.

In essence, the problem consists on efficiently ranking the URLs clicked by previous users that submitted similar queries to the search engine. For this purpose, clicks themselves are indexed and now concurrency conflicts appear among the continuous stream of click updates over the index and the required operations needed to process time-consuming tasks such as determination of similar queries and related clicks on documents.

By "similar" we mean queries correlated in some probabilistic way that considers clicked URLs and respective query terms. The calculation of the probabilities query-to-query, query-to-URL, and URL-to-URL can be very demanding in execution time and memory requirements with the additional challenge that must be performed on-the-fly for each user query.

In this paper we present algorithms for dealing with the problem of on-line indexing and querying a continuous stream of queries generated by users of a large Web search engine. We concentrate on what happens on a multicore-based click-ranking node dealing with this work-load and in particular we focus on how to organize the associated concurrent read/write operations submitted from the different threads running on such a click-ranking node. Our main contribution is a comparative study of different strategies that trade-off parallelism granularity and data locality.

The remaining of the paper is structured as follows. In Section 2 we provide some background and discuss related work. In Section 3 we describe different strategies explored by this research. In Section 4 we analyze our experimental results and in Section 5 we conclude summarizing our findings.

2 Background and Problem Setting

Web Search Engines use the inverted file data structure to index the text collection and speed up query processing. An inverted file is composed of a vocabulary table and a set of posting lists. The vocabulary table contains the set of relevant terms found in the collection. Each of these terms is associated with a posting list which contains the document identifiers where the term appears in the collection along with additional data used for ranking purposes. To solve a query, it is necessary to get the set of documents *ids* associated with the query terms and then perform a ranking of these documents so as to select the top K documents as the query answer. On conventional search engines, the posting list are update off-line and consequently query operations are exclusively read-only requests upon the inverted file.

A number of papers have been published reporting experiments and proposals for efficient parallel query processing upon inverted files which are distributed on a set of P **nodes** [1–3, 16, 15, 19, 17]. The two dominant approaches to distributing an inverted file are (a) the **document partitioning** strategy (also called local indexing), in which the documents are evenly distributed onto the

set of available nodes 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 onto the processors. Document partitioning is usually employed since it has better scalability. These strategies have been devised for distributed memory systems in which nodes have a share nothing architecture. Note, however, that their main principles can also be re-used on a multi-core setting in which nodes consists of several cores interconnected through a shared memory hierarchy.

Our context differs from a conventional set-up in several key aspects.

- First, our clicks engine has an inverted file that indexes URLs clicked by users in previous actual queries submitted to the Web search engine. The vocabulary table of that inverted file is formed by the query terms – or queries themselves treated as single units – and the associated postings lists contain references to the URLs clicked by users together with other data used for ranking. Furthermore, as shown in Figure 1 we need double inverted indexing so that from terms we can reach clicked URLs and vice-versa. A key operation is to start from the query term to reach a set of clicked URL, then these URLs are used to get a new set of terms from the second inverted file and from these terms get more URLs. The resulting sets of terms and URLs are then operated each other to generate a list of ranked URLs.
- Second, by "query" we mean a number of operations performed on the inverted file that tracks the relevance of the click made by users and this means that we need to process both read and write requests upon the inverted file. As mentioned above, in conventional search engines queries are usually read-only requests since the update of the posting list is performed off-line. However, for real-time indexing of click-through data, it is mandatory to process on-line write requests upon the inverted file to keep posting list up-to-date all the time. As new queries arrive to each click-ranking node it is necessary to detect if the clicked URLs are already being indexed. If so, the clicks count of the respective URLs must increased and the item promoted to the front of the posting lists associated with the query terms. We use a transposition heuristic on the posting list to promote highly clicked URLs to the front of the posting list. This is used as a low cost indication of how recently the URLs have been clicked which is useful for ranking purposes.
- Third, our engine prototype is explicitly designed to exploit the available thread level parallelism available in current multicore processors. A straightforward approach to transparently take advantage of such architectures is to rely on virtualization technology and use as many single-CPU virtual nodes per processor as cores are available. Unfortunately, this involve additional overheads that become an overkill in the extremely demanding arena of search engines. Note, however, that explicit parallelism complicated click-ranking node design and implementation, especially in this setting with concurrent read and white request upon the inverted file.

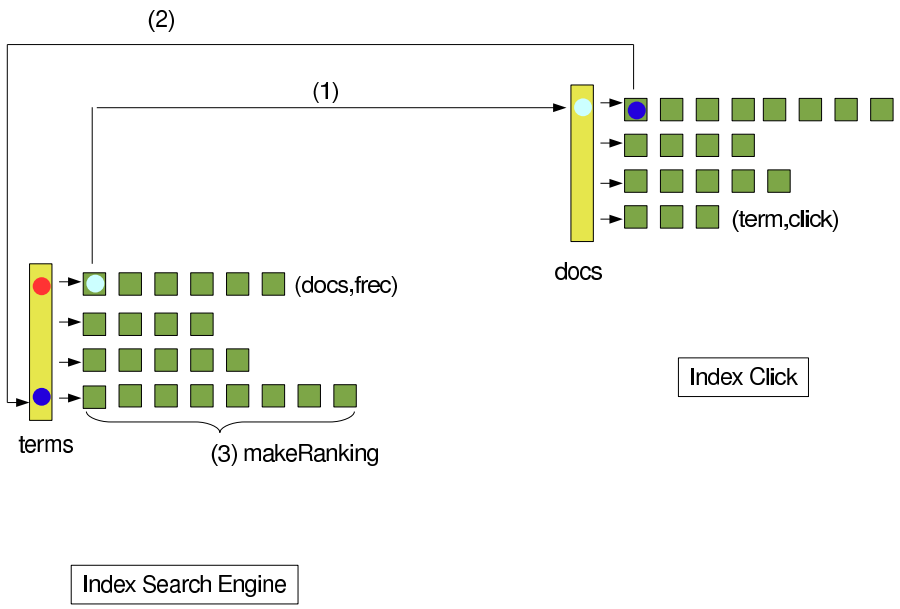

Fig. 1. Double inverted file organization. The first index (left) is a standard search engine index in which *doc* is a clicked URL and *frec* is the total number of times users have clicked the URL for queries containing the same term. The second index (right) enables the mapping from clicked URLs to query terms that caused the clicks on the URLs, where *click* indicates the average position in the result webpage of the respective URL. The sequence given by the labels (1), (2) and (3) indicates that from a given term (1) it is possible to reach a new term (2), which in turn leads to a new set of documents (3) to be included in the ranking process. For each posting list item in the first index, this sequence is repeated for each posting list item of the second index. Therefore, queries expand the set of active terms during a period of time and new query arrivals cause the modification of the posting lists of both indexes which potentially causes read/write conflicts.

Our focus in this paper is to explore different alternatives to implement such a parallel click-ranking node. Intuitively, the simplest approach consists in exploiting thread level parallelism at the query level, i.e. assigning an independent thread per incoming query. This approach could perform well from a parallel implementation point of view as long as there are always enough simultaneous queries to keep all cores busy. In fact, this is the interesting case since performance only becomes critical when the engine operates under heavy query traffic. When traffic is sufficiently low, it does not really matter that a given strategy is less efficient than another provided that the response time of individual operations is below an upper limit. However, even assuming peak traffic, if we need to update ranking information online based on clicks made by users, read

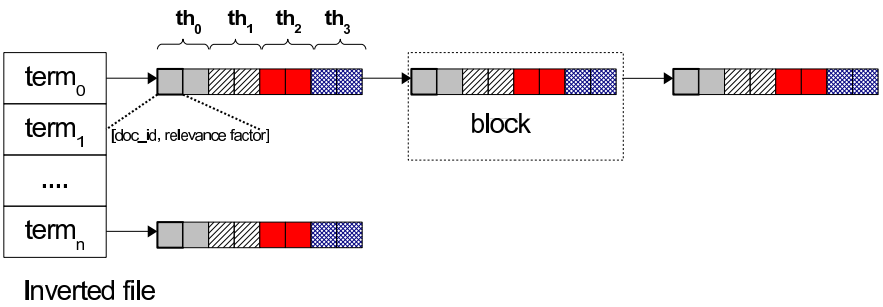

Fig. 2. Inverted file organization where each posting list is stored in a number of blocks and where each block is logically divided in chunks that are assigned to threads. Each chunk is composed of a number of posting list items and each item is a pair (doc_id, relevance factor). In this example, the first block of *term_0* is processed in parallel by threads *th0*, *th1*, *th2* and *th3*.

writer synchronization may jeopardize parallel performance and it is unclear if exploiting query-level parallelism will be enough.

As an alternative we need to evaluate if the processing of a single query itself can be organized to exploit intra-query parallelism. Intuitively, this is also possible since the posting list of the inverted file are large enough and data parallelism can be exploited when traversing such lists. This is the idea illustrated graphically in Figure 2. Items on the posting lists (URL references) are usually kept ordered by a relevance factor that accounts for the frequency of clicks made by users to them, but for efficiency reasons, instead of keeping a fully ordered list, we group items into different blocks and kept the list sorted by the relevance factor just across blocks, i.e. the set of items kept in block i are all of higher relevance than the values associated with the set of items kept in block $i + 1$. With this organization, the idea is that posting list processing can be perform on a block basis and within each block, we can exploit parallelism at item level distributing each block into chunks with are assigned to the available threads.

In summary, in this paper we have tried to answer to the following complementary questions:

1. Is query-level parallelism enough to achieve satisfactory parallel performance under heavy query traffic?
2. If query-level parallelism is not enough, is it efficient to exploit parallelism at the item level?
3. How to implement the concurrency control mechanism required by the read write synchronization inherent to the online update?

3 Strategies for Read-Write Synchronization

As mentioned above, we have studied a number of strategies for implementing real-time parallel indexing of click-though data which (1) exploit parallelism at different levels (either query level or item level or both) and (2) implement different concurrency control policies to satisfy read write synchronization issues. All of them use either locks or barriers as synchronization mechanism and their main characteristics are summarized in Table 1.

The most restrictive strategies guarantee that the scheduling of regular searches on the inverted file, which we denoted as read transactions, and ranking update operations, which we denote as write transactions, are serializable, i.e. they maintain the illusion of serial execution. The other strategies does not enforced serialization, which potentially will allow for better performance. Nevertheless, the proposed strategies make use of data locality in different ways and this also have a strong influence on performance.

The "**Bulk Processing (BP)**" strategy does not exploit parallelism at the query level, i.e. either regular searches on the inverted file or updates of the posting lists, are processes serially without any kind of concurrency. Instead, it exploits parallelism a much finer level. As described above, the posting lists are statically divided into blocks. When serving read transactions, these blocks are logically partitioned into smaller chunks, which are processed in parallel by the available threads within the click-ranking node. A master thread performs a short sequential phase to merge the results of those threads and finally it gets the same outcome as a conventional single CPU algorithm. The block size is a key parameter in this approach. Intuitively, the larger the block size, the coarser the parallelism, but smaller blocks tend to improve data locality and some trade-off should be found.

The "**Concurrent-Reads (CR)**" does exploit query level parallelism but just for regular searches. However, before a write operation could take place, the CR strategy waits for all of the current reads being solved to end. In this way, read transactions exploit the available (intuitive) parallelism between independent queries but as in BP, write serialization is also guaranteed by isolating the execution of write transactions. In fact, write transactions are handle the same way in both approaches.

The "**Term-Level-Parallelism (TLP)**" strategy allows concurrency of both read and write operations as long as they involve different terms of the vocabulary and they are not correlated. Concurrent transactions are assigned on demand to the available threads of the click-ranking node and ideally, a different lock protects the posting list of each vocabulary term to control the concurrency. To enforce serializability, a thread does not proceed with a transaction till it acquires all its associated locks at once. In practice, since locks are an expensive resource, this approach uses hashing to map several terms onto the same lock. We have explored two alternative variations. The first one always allow concurrent reads (*TLP1*), whereas the second, which is much easier to implement, does not overlap read transactions that have some terms in common (*TLP2*).

Finally, we have also explored a couple of strategies which relax serializability requirements. The first one, which is denoted as "**Relaxed-Term-Level-Parallelism (RTLP)**", is similar to the TLP approach but without forcing threads to acquire all the locks of a given transaction at once and removing the atomic commitment of transactions. Our second approach, which we have denoted as "**Relaxed-Block-Level-Parallelism (RBLP)**", uses a similar strategy but controls concurrency at the block level (a hashing function maps several blocks of the posting lists to the same lock) to reduce potential imbalances caused by the Zipf Law. Obviously, the degree of concurrency of both RTLP and RBLP is much higher than the other strategies but they may introduce a non-serializable scheduling. Nevertheless, in click-ranking it usually does not matter much if some inconsistencies appear.

Table 1. Different strategies for implementing a multicore-based click-ranking node supporting online updates of the inverted index. The first column indicates if the strategy introduces non-serializable scheduling of read and write transactions, the second column indicates the source of parallelism and the third column indicates the inner mechanism used to control concurrency and enable online updates.

	Serializable	**Parallelism**	**Implementation**
BP	Yes	Fine Grain	Barrier based
CR	Yes	Concurrent Searches	Barrier based
TLP1	Yes	Concurrent Searches and Updates	Term Lock based
TLP2	Yes	Concurrent Searches and Updates	Term Lock based
RTLP	No	Concurrent Searches and Updates	Term Lock based
RBLP	No	Concurrent Searches and Updates	Block Lock based

4 Experiments

The computing platform used in our simulations is a dedicated cluster node equipped with two *Intel's Quad-Xeon* processors, whose main characteristics are summarized in Table 2. The search node prototype have been developed in C++, using Linux *POSIX Threads* as explicit threading API.

Our evaluation has focused on studying the performance under heavy query traffic since the ultimate goal is to deploy index nodes able to efficiently cope with drastic peaks in traffic. In particular, we have evaluated two different scenarios that emulated extreme cases:

1. **Workload 1** – Heavy traffic but limited concurrency –. In the most adverse scenario we have evaluated, there is a high probability that subsequent queries (in time) actually become quite similar from a semantic point of view. For instance, this emulates traffic when suddenly, many people becomes interested on the same topic and the engine receives a large stream of similar

Table 2. Main features of the target computing platform.

Processor	Intel Quad-Xeon (2.66 GHz)	
	L1 Cache (per core)	4x32KB+4x32KB (inst.+data) 8-way assoc. 64 byte/line
	L2 Unified Cache	2x4MB 16-way assoc. 64-byte/line
Memory	16 GBytes (4x4GB) 667 MHz FB-DIMM memory 1333 MHz system bus	
Operating System	GNU Debian System Linux kernel 2.6.22-SMP for 64 bits	
Intel C/C++ v.10.1 Compiler Switches	-fast Parallelization with POSIX Threads: -lpthreads	

queries. In this case ranking updates and regular accesses to the same posting lists (i.e. read and write transactions) occur almost simultaneously very often, which limits the available parallelism of some of the strategies.

2. **Workload 2** – Heavy traffic, high concurrency –. This workload emulates a more benign scenario in which the chances of simultaneous read and write transactions to the same posting lists is much lower, which can be considered as an average case. Since subsequent queries are less correlated, there is a much coarser parallelism and it is expected that relaxed schemes can be benefited from it.

Simulations have been performed with a text collection from an United Kingdom Web database sample. Query traces have been build using a real query-log containing user searches, which in turn, have been used to build synthetic click-through traces using different user behavior models. Obviously, users do not click on links at random, but make an informed choice that depends on many factors, but for the focus of this paper, we believe that this random model is enough to obtain useful insights on the comparative performance of the strategies tested in this work. Nevertheless, we are aware of our simplifications and it is clear that precise modeling of click-through behavior or real click-logs will be necessary to refine our experiments.

In our experiments we have assumed than users click on average just on one of the most promising top-k links presented in the search results page, but we have also evaluated more demanding workloads in which for each search query users clicks on more than one link and hence more ranking updates are necessary for a single search request.

There are many parameters that have a noticeable influence on the performance of our click-ranking node but for the sake of clarity we only summarize here our major findings and report results using optimal parameters.

One of the most important parameters is the block size used within the posting lists. Intuitively, the larger the block size, the coarser the parallelism for

strategies like BP, but smaller blocks tend to improve data locality so a trade-off is in place. Empirically, we have found that blocks of 128 elements provide the best performance rates across all strategies and this is the block size used in all our experiments.

Another important parameter is the number of locks used in strategies such as *TLP* and *RTLP*. As locks resources are limited and their administration is expensive in terms on running time, the strategies TLP1, TLP2 and RTLP use hashing to reduce the number of actual locks created by the program. To assess the impact of hash collisions in this application setting, we tried with different number of locks and fortunately, we found that even with a moderate number of identifiers, namely an array of locks (hash table) with no less than 4099 entries, the performance degradation over having unlimited locks becomes negligible (less than 2%).

Performance Results

Figure 3 shows the scalability of the different strategies using the workload 1. Surprisingly, BP is the strategy that scale the best in our platform and reaches an impressive speedup of 7.75x for eight threads, in spite of only exploiting what we have denoted above as item level parallelism. This is an important finding since intuitively, the other approaches are able to exploit coarser parallelism. However, click-ranking is a demanding application since read-write synchronization is very frequent and in practice, even if we are assuming high query traffic, query level parallelism is not enough to keep many concurrent threads busy. Figure 4 further demonstrate this issue. If we assume that users click on average on to four link for each search, more ranking updates will be needed and less query level parallelism will be actually available.

Figures 5 and 6 focused on 8-thread experiments and show the actual query throughput *and* index-update throughput achieved with workloads 1 and 2 respectively. Under a less demanding workload (Figure 6), query level parallelism is higher, but even in that case, BP is able to outperform the other strategies since read-write synchronization overheads are still large enough.

Finally, Figure 7 shows running times of individual query and index-update operations. However, these results has to be read with precaution since they measure the time elapsed between the instant in which the query/index-update operation starts execution and the instant it is completely processed. Precision of measures can be compromised given the tiny values of running times. However, they show a general trend from which we can explain the differences in overall query throughput observed in the above experiments. First, the curves in Figure 7 [top] show that average running time per operation tends to be very similar each other across strategies. They are smaller for 5 clicks indicating that index-update operations are faster than query operations and the trace executed in that case is populated by more of these faster operations. Except for the BP strategy, all other strategies basically assign one thread to process sequentially the query/index-update operation. One would expect BP to outperform all others in this case because it uses all threads to process each query/index-update

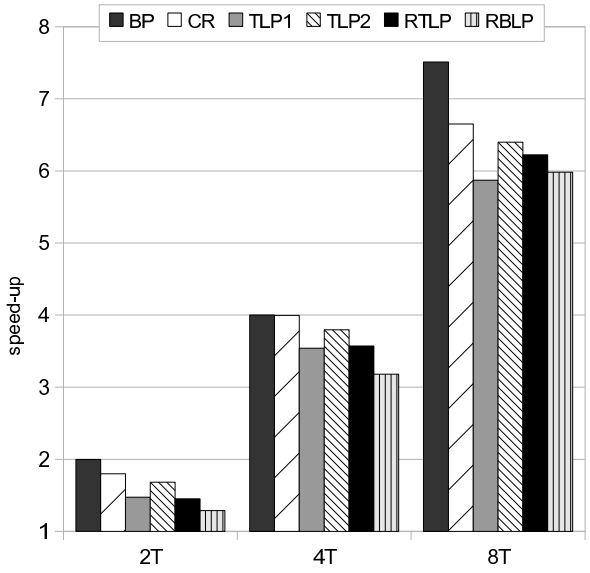

Fig. 3. Scalability of the different approaches for workload 1 and assuming users just click on one link per search on average. Results for 2, 4 and 8 threads (T).

operation. However, BP has the burden of barrier synchronizing the threads in order to start with the next operation, and the results show that this cost is significant. On the other hand, the points in Figure 7 [bottom] clearly show that all other strategies tend to consume a significant amount of time for some operations which is an indication that they suffer, from time to time, from long delays due to lock contention among the active threads. The BP strategy does not suffer from this problem because it simply processes one operation at a time and, while it uses locks for implementing oblivious barrier synchronization, the threads do not have to compete each other to acquire locks at the end of each operation. This explains the better performance of BP with respect to the relevant performance metric for our case, namely query/index-update throughput.

5 Conclusions

We have presented an experimental study that compares different concurrency control strategies devised to process user clicks in an on-line manner. The results show that the BP approach based on exploiting the full parallelism available from the cores for processing one single query or index update at a time, is the best alternative. This strategy outperforms the more intuitive approach found out as current practice in multi-threaded search engine nodes. Namely, the strategy in which each active query or index update is handled by an independent concurrent thread that is mapped to one of the available cores. We tested different variants

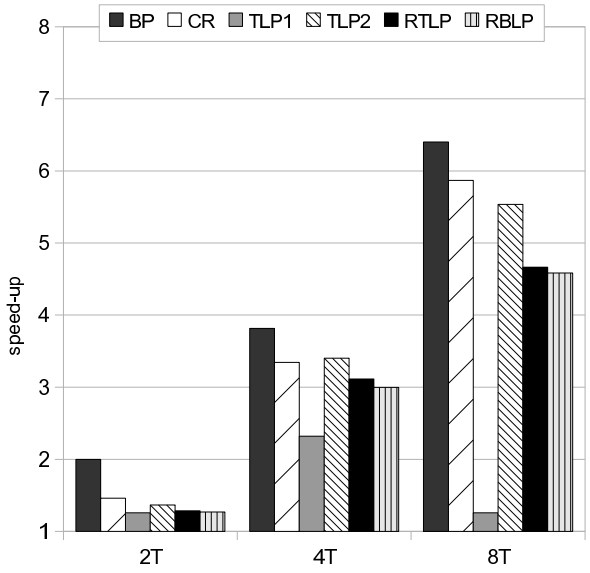

Fig. 4. Scalability of the different approaches for workload 1 and assuming a more demanding experiments in which users click on four links per search on average. Results for 2, 4 and 8 threads (T).

of the intuitive approach. Each one representing differing degrees of compromise between fully concurrent operation and strict serialization of read/write transactions. The results show that even for the very relaxed strategies, the BP approach (which is serializable) performs better.

References

1. A. Arusu, J. Cho, H. Garcia-Molina, A. Paepcke and S. Raghavan. Searching the web. In *ACM Trans* 1(1):2–43, 2001.
2. C. Badue, R. Baeza-Yates, B. Ribeiro, and N. Ziviani. Distributed query processing using partitioned inverted files. In *SPIRE*, pp. 10-20, 2001 (IEEE-CS).
3. A. Barroso, J. Dean and U. H. Olzle. Web search for a planet: The google cluster architecture. In *IEEE Micro* 23(2): 22–28, 2002.
4. R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. Design trade-offs for search engine caching. ACM TWEB, 2(4):1–28, 2008.
5. R. Baeza and B. Ribeiro, Modern Information Retrieval, Addison-Wesley.1999
6. S. Ding, J. He, H. Yan and T. Suel. Using Graphics Processors for High Performance IR Query Processing. In
em WWW, pp. 421–430,2009
7. K. Dragicevic and D. Bauer. A survey of concurrent priority queue algorithms. In *IPDPS*, pp. 1–6, 2008.

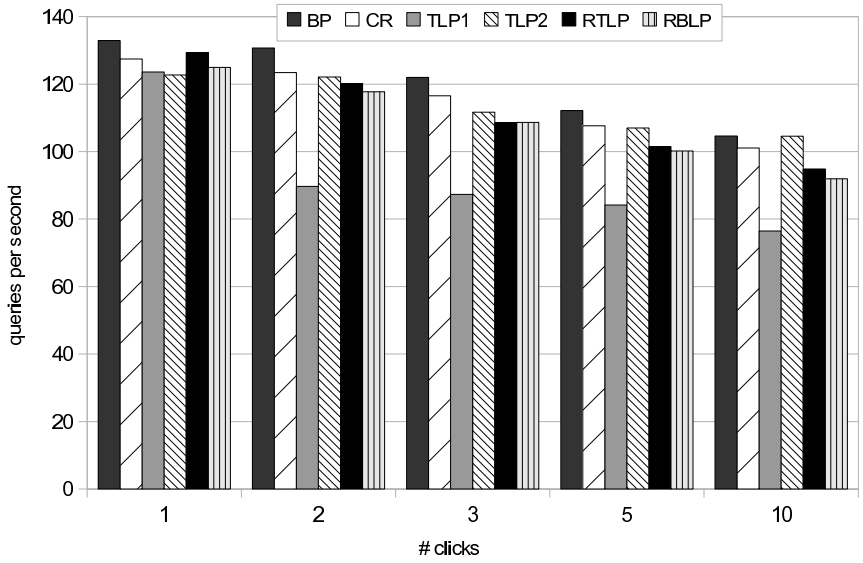

Fig. 5. Query/index-update throughput achieved with workload 1 under different click-through behavior from users. Results for 8 threads.

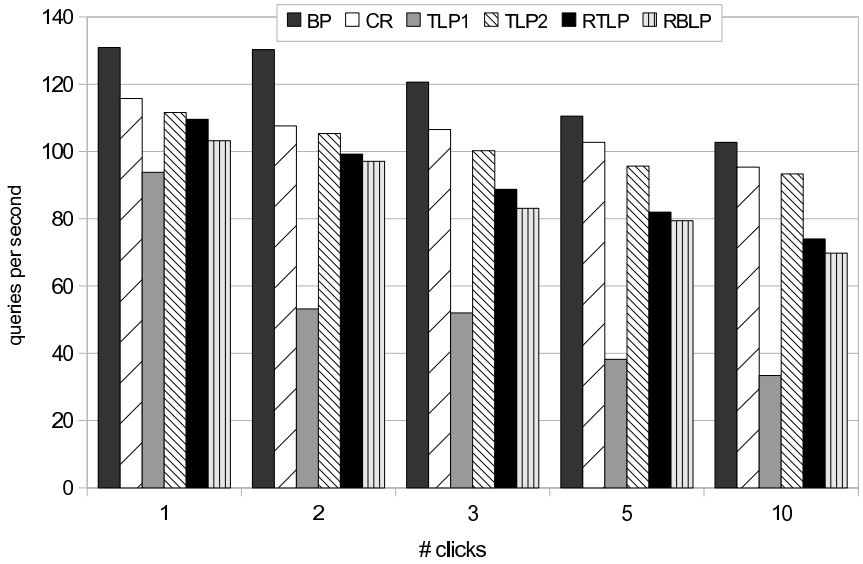

Fig. 6. Query/index-update achieved with workload 2 under different click-through behavior from users. Results for 8 threads.

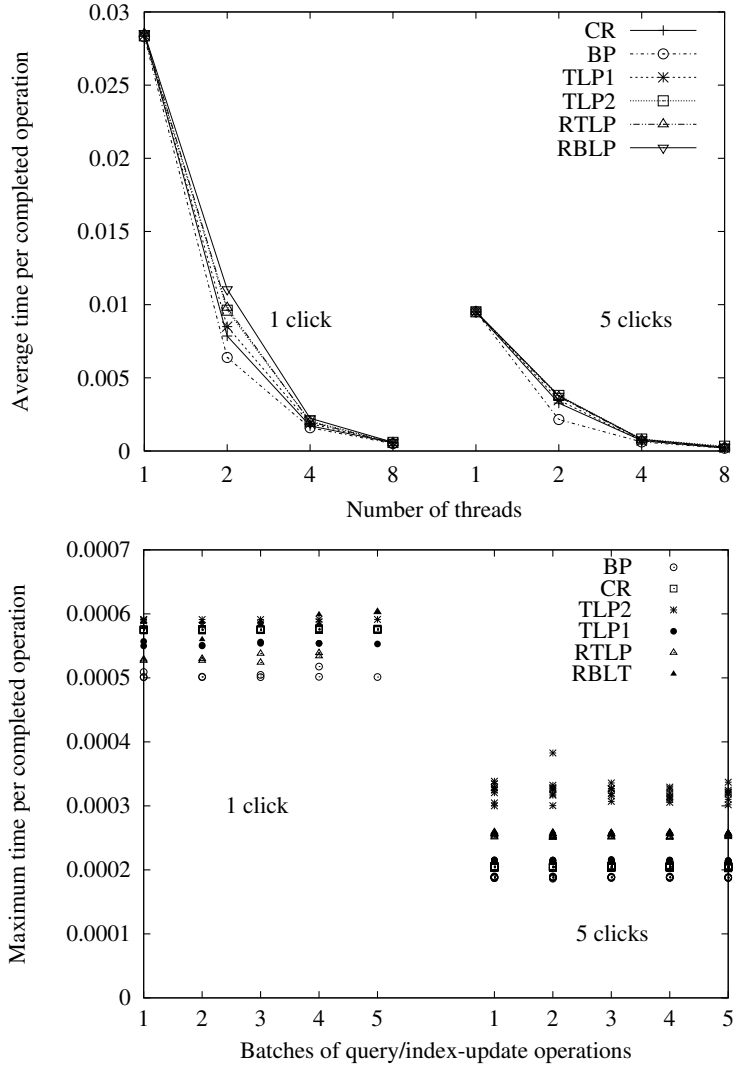

Fig. 7. Individual query times. The figure at the top shows results for average running time per query/update-index operation for 1, 2, 4 and 8 threads. The figure at the bottom shows the maximum running time observed every 1,000 query/update-index operations for 8 threads.

8. T. Fagni, R. Perego, F. Silvestri, and S. Orlando. Boosting the performance of Web search engines: Caching and prefetching query results by exploiting historical usage data. ACM TOIS, 24(1):51-78, 2006.
9. Q. Gan, T. Suel. Improved Techniques for Result Caching in Web Search Engines. In *WWW*, pp. 431–440, 2009.
10. B. S. Jeong and E. Omiecinski. Inverted file partitioning schemes in multiple disk systems. *IEEE Trans. Parallel and Distributed Systems*, 16(2):142–153, 1995.

11. R. Lempel, and S. Moran. Predictive caching and prefetching of query results in search engines. In *WWW*, pp. 19–28, 2003.
12. X. Long, and T. Suel. Three-level caching for efficient query processing in large Web search engines. In 14th *WWW*, pp. 257-266, 2005.
13. A. MacFarlane, J. McCann, and S. Robertson. Parallel search using partitioned inverted files. In *SPIRE'02*, pp 209–220., 2000 (IEEE CS).
14. E. Markatos. On caching search engine query results. Computer Communications, 24(7):137-143, 2000.
15. M. Marin and V. Gil-Costa. High-performance distributed inverted files. In Proc. *CIKM* pp. 935–938, 2007.
16. M. Marin, C. Bonacic, V. Gil-Costa and C. Gomez-Pantoja. A Search Engine Accepting On-Line Updates. In *Euro-Par '07: 13th International Conference on Parallel and Distributed Computing* LNCS 4641 pp. 348–357, 2007.
17. J. Zobel and A. Moffat. Inverted Files for Text Search Engines. In *ACM Computing Surveys* 38(2). 2006
18. M. Marin, R. Paredes and C. Bonacic, High-Performance Priority Queues for Parallel Crawlers, In *10th ACM International Workshop on Web Information and Data Management (WIDM 2008)*, California, US, Oct. 30, 2008.
19. W. Moffat, J. Webber, Zobel, and R. Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Information Retrieval*, Aug. 2007.
20. M. Persin, J. Zobel and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. In *Journal of the American Society for Information Science*, 47(10): pp 749–764, 1996.
21. B. Ribeiro-Neto and R. Barbosa. Query performance for tightly coupled distributed digital libraries. *ACM Conference on Digital Libraries*, pages 182–190, 1998.
22. W. Xi, O. Sornil, M. Luo, and E. A. Fox. Hybrid partition inverted files: Experimental validation. In *ECDL'02*, pp. 422–431, 2002.