Exploiting Query Term Correlation for List Caching in Web Search Engines

Jiancong Tong[†],[*] Gang Wang[†] Douglas S. Stones[‡§] Shizhao Sun[†], Xiaoguang Liu[†], Fan Zhang[†]

[†]Nankai-Baidu Joint Lab, Nankai University, Tianjin, 300071, China
[‡]School of Mathematical Sciences, Monash University, VIC, 3800, Australia
[§]Department of Mathematics and Statistics, Dalhousie University, Halifax, NS B3H 4H8, Canada

ABSTRACT

Caching technologies have been widely employed to boost the performance of Web search engines. Motivated by the correlation between terms in query logs from a commercial search engine, we explore the idea of a caching scheme based on pairs of terms, rather than individual terms (which is the typical approach used by search engines today). We propose an inverted list caching policy, based on the Least Recently Used method, in which the co-occurring correlation between terms in the query stream is accounted for when deciding on which terms to keep in the cache. We consider not only the term co-occurrence within the same query but also the co-occurrence between separate queries. Experimental results show that the proposed approach can improve not only the cache hit ratio but also the overall throughput of the system when compared to existing list caching algorithms.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Search process*

Keywords

Search engines, list caching, term co-occurrence, query logs

1. INTRODUCTION

Popular Web search engines receive millions of queries daily, and users have high expectations of the quality and speed of the answers. Caching technologies are widely employed by Web search engines, as they can be used to reduce the amount of computation and disk access required, thereby resulting in shorter response times and higher throughput.

Caching in Web search engines has been studied mostly on two levels [9, 12]: *result caching* and *list caching*. Result caching is to store in memory the list of documents associated with a given query. If the same query has been recently submitted repeatedly, the system can simply return the cached result instead of re-evaluating the entire query. List caching is to keep the inverted lists of frequently used terms in main memory. It will reduce the disk data transfers when processing new queries that comprise at least one of those terms. These two approaches are often combined to achieve better performance [2, 11].

These caching methods can be further classified according to if the decision of cache content is made off-line (*static*) or on-line (*dynamic*) [1, 10]. Static caching usually exploits historical data and caches the most frequently accessed items. The items are kept in memory in advance and would not be replaced during the query processing. Dynamic caching stores the most recently accessed items in memory, and updates the content "on the fly" as queries are submitted.

In this paper we focus on dynamic list caching. Well-known dynamic list caching policies include Least Recently Used (LRU) [10], Least Frequently Used (LFU) [14], and the state-of-the-art QtfDf policy [1]. Each of these methods focuses on how to evict the "stale" items in order to make enough room for new ones. However, a query cannot be immediately evaluated as long as one of its terms has not been cached (regardless of whether or not all the other terms have been cached). This motivates the study of a caching policy that accounts for which terms appear together in queries.

In this paper, we present a list caching approach, which takes advantage of the co-occurrence of query terms. More specifically, when it comes to decide which items to be admitted or evicted, the novel caching policy considers not only the temporal and spatial locality of terms in a data set, but also the correlation among the terms. The experimental results demonstrate that the proposed list caching policy (referred as QTCA, abbreviated for Query Term Correlation Aware) can improve both the hit rate and the overall query throughput.

2. RELATED WORK

The problem of caching inverted lists, has been studied extensively (see e.g. [1, 2, 3, 4, 9, 11, 12, 14]). For this paper, we will compare our proposed method with the classic LRU and LFU policies, as well as the QtfDf policy proposed by Baeza-Yates et al. [1] which outperformed all the previous methods. In [1], the QtfDf strategy considers the ratio of query-term frequency (the number of queries containing the term in the query log) to the length of the inverted list (the number of documents containing the term in the query log) when admitting and evicting items. Note that in this paper, only the dynamic version of QtfDf is used as a baseline.

[*]Email: lingfenghx@gmail.com

CIKM'13, October 27 - November 01 2013, San Francisco, CA, USA.

A dynamic caching strategy usually consists of both an admission policy and an eviction policy. Admission policies have been studied in [3, 14] independent of any cache replacement policies. These methods aimed at picking the most-frequent items to cache in order to maximize the benefit of the cache. Meanwhile, eviction policies [8, 9] focus on identifying the cached items that are least likely to re-occur and replacing them with the ones selected by the admission policies. Unlike previous work, our caching policy exploits the term correlation, instead of the stereotyped temporal and/or spatial locality pattern, to determine which items should be admitted or evicted.

The term correlation in query logs has already been studied in previous work. Silverstein and Henzinger [13] presented a correlation analysis of query logs and studied the interaction of terms within queries; the results suggested that it might be useful for search engines to consider query terms as parts of phrases. Chaudhuri et al. [6] observed that the distribution of combinations of multiple terms in query logs exhibited power law behavior, while Chau et al. [5] confirmed that the distribution of n-gram term combinations also followed the Zipfian distribution (a special kind of power law distribution). However, to the best of our knowledge, correlation between pairs of terms has not been utilized in the context of list caching in search engines.

3. MOTIVATION

The motivation of our novel policy comes from the observation that the frequency distribution of term pairs in the query log of real-world search engine approximately follows a power law distribution. This property has been previously observed by e.g. [1, 5, 6], and we will observe it is also true for the data used in this paper.

3.1 Query Log

In this study, our query trace contains 1.5 million queries which are randomly sampled from a large log of queries submitted to a commercial search engine over three months. Out of the 4.95 million query terms, there are 154,855 distinct terms, with the most frequent term appearing 78,617 times. The plot marked *term* in Figure 1 gives the frequency distribution of the top 10^5 ranked terms on a log-log scale. (We will later compare this plot to similar plots for term pairs.) As expected, the query term frequencies follow an approximate power law distribution, with slope of 1.37.

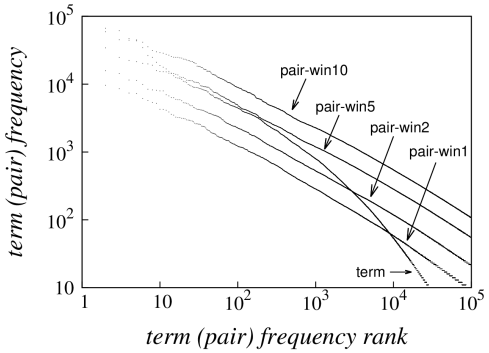

Figure 1: Frequency distribution of query terms and co-occurring pairs (win$X$ denotes a window size X)

3.2 Co-occurring Term Pairs

We describe two distinct terms as *co-occurring* if they occur within queries spaced at most w apart. Note that this includes the possibility that they occur within the same query. Here we call the union of several consecutive queries as the *window* whose size is w. Within a given window, any ordered pair of two co-occurring terms are called co-occurring pair. A more formal definition of co-occurring pair is provided in Section 4. For example, if we have the three queries

$$\text{“}A\ B\text{”}, \qquad \text{“}C\text{”}, \qquad \text{“}E\ B\text{”}$$

in succession, and $w = 2$, then this gives rise to the co-occurring pairs:

$$(A, B), \quad (A, C), \quad (B, C), \quad (C, E), \quad (C, B), \quad (E, B).$$

Figure 1 also plots the frequency distribution of the top 10^5 ranked co-occurring pairs. There are four plots (marked *pair-win$X$*, where $X = w$) corresponding to four window sizes $w \in \{1, 2, 5, 10\}$. As with *term* in Figure 1, a power law relationship appears to holds for the distribution of term pairs within queries, and similarly for term pairs within a range of window sizes. The slopes of the four curves are in the range of 0.7 to 0.75.

The observation suggests that some terms tend to appear together within a certain *window* and a small number of co-occurring term pairs are significantly more common than others. This motivates the idea that it is better to keep both terms (of a co-occurring pair) in the cache space, using fetch-ahead and deferred-eviction methods, which leads to our query term correlation aware caching policy.

4. PRESENTING QTCA CACHING

In this section, we describe a novel list caching policy, the *QTCA* (Query Term Correlation Aware) method, which exploits the correlation among the query terms.

Let $q_1, q_2, \ldots, q_n$ be a consecutive sequence of queries. Given two distinct terms u and v, a term pair (u, v) is described as *co-occurring* if $u \in q_i$, $v \in q_{i+k}$ and $0 \le k < w$ (and if $k = 0$ we have u before v in q_i). Here $w \ge 1$ is the *window size*. If (u, v) is a co-occurring term pair, we say that v is the *associated term* of u. For example, in the example in Section 3.2, the associated terms of A are B, C, and D.

QTCA consists of two major components: (i) An algorithm for mining, from previous query logs, the term pairs that frequently co-occur. We consider both *intra-query* term pairs (i.e., within a single query) or *inter-query* term pairs (i.e., within a window of queries) in the past query logs. (ii) A caching strategy, a LRU-based policy, which takes both term dependence and co-occurrence into consideration.

4.1 Mining the Correlation

Algorithm 1 presents the routine for mining co-occurring pairs, while the parameters used are described below:

- We do not include any term that occurs fewer than *Sup* times, the *minimum support*; these are discarded at line 1 of Algorithm 1. We also do not include any term pair that occurs fewer than *Sup* times (see line 13).
- Let u and v be two distinct terms and S be a co-occurring pair (u, v). Let $N(u)$ denote the number of times term u appears in queries, and let $N(S)$ denote the number of times terms u and v co-appear in queries. Define

  $$P(u) = \frac{N(u)}{\sum_a N(a)} \qquad \text{and} \qquad P(u, v) = \frac{N(S)}{\sum_A N(A)}$$

  where the sums are over all terms a and all sets of two distinct terms A, respectively.

- We define *Conf* as the *minimum confidence level* and let $d(u,v) = \frac{P(u,v)}{P(u)}$; term pairs (u,v) that do not satisfy $d(u,v) \geq Conf$ are discarded (see line 13). For practical reasons, the definition of *Conf* here is slightly different from the typical definition used in frequent pattern mining. The number of unique term pairs is so large that it is impossible to store all candidate pairs in memory, therefore we divide the unique terms into several parts (*Cut* is equal to the number of partitions) and process one part each time. Thus the condition $d(u,v) \geq Conf$ applies to each part individually.
- We use *Win* to denote the size of the sliding window (previously denoted w). We can restrict to intra-query term pairs that can be obtained by setting $Win = 1$.

Algorithm 1 Co-occurring term pair mining

Input: Query log $\mathcal{Q}$, Parameters *Sup*, *Conf*, *Win*, *Cut*
Output: A set $\mathcal{S}$ of co-occurring term pairs

```
1: Generate a set T of terms from Q, and discard the terms
   that occur fewer than Sup times in Q
2: Divide T into subsets T_1, T_2 ... T_Cut
3: S ← ∅
4: for k = 1, 2, ..., Cut do
5:     Create a 2D array F = f[u, v]
6:     f[u, v] ← 0 for all u ∈ T_k, v ∈ T
7:     Scan Q with window size Win to identify term pairs
8:     for each term pair (u, v) identified do
9:         if u ∈ T_k and v ∈ T then
10:            f[u, v] ← f[u, v] + 1
11:        end if
12:    end for
13:    S ← S ∪ {(u, v) : f[u, v] ≥ Sup and d(u, v) ≥ Conf}
14: end for
15: return S
```

4.2 QTCA Caching Strategy

A dynamic caching strategy usually consists of both an admission policy and an eviction policy. However, most of the existing caching policies are actually cache eviction policies, and do not pay much attention to the admission policy, simply admitting nothing other than (some of the) incoming query terms. The QTCA policy proposed here adopts a more complicated admission rule where some terms are fetched into cache in advance (before it is requested) due to its associated terms. This is called *associated fetching*.

Algorithm 2 describes the proposed QTCA caching strategy. Here $\ell(t)$ denotes the posting list of term t and B denotes the cache buffer, which contains the cached posting lists $\ell(t)$. If a list $\ell(t)$ has already been cached, and the *associated fetching condition* of t is satisfied, we add $\ell(p)$ to the cache, for all terms p that co-occurred with t (line 2 of Procedure `ASSOFETCH`). The associated fetching condition can be defined in various ways. One reasonable condition is to check if the accessed times of t exceeds a given threshold n (we set $n = 2$ in our tests) since it has been admitted in the cache. Another option is to check if t has resided in the cache for time z or longer.

We wish to highlight that the associated fetching should not interfere the normal query processing, since they can be performed by asynchronous read operations. When a query is submitted to the search engine, the non-cached posting lists will be fetched from the disk first and then the query is evaluated. The associated terms will not be prefetched until the "on-the-fly" fetchings have been completed. Thus the search engine will never be blocked on answering a query due to the prefetch of associated terms.

Unlike typical eviction policies, the eviction policy in QTCA also accounts for term pair correlation. The algorithm begins by picking an eviction candidate s (lines 3 in Procedure `MAKEROOMFOR`) as per the usual LRU method. However, $\ell(s)$ is not immediately evicted. Instead, it is given a second chance to "survive" provided it has at least one associated term in the cache ($Chance_s$ is the corresponding flag and is set as 0 when $\ell(s)$ is added to the cache buffer).

Algorithm 2 Term Correlation Aware Caching Strategy

Input: Query term t, Set of co-occurring term pairs $\mathcal{S}$, Cache Buffer $\mathcal{B}$

```
1: procedure ADMIT(t)                          ▷ admission policy
2:     if ℓ(t) has already been cached then
3:         ASSOFETCH(t)
4:     else
5:         MAKEROOMFOR(t)
6:         Admit ℓ(t) into B
7:     end if
8: end procedure

1: procedure ASSOFETCH(t)                      ▷ associated fetching
2:     if the AssoFetch condition of t is satisfied then
3:         P ← {p : (t, p) ∈ S ∧ ℓ(p) is not cached}
4:         for each term p ∈ P do
5:             MAKEROOMFOR(p)
6:             Admit ℓ(p) into B
7:         end for
8:     end if
9: end procedure

1: procedure MAKEROOMFOR(t)                    ▷ eviction policy
2:     while there is not enough space in B for ℓ(t) do
3:         Pick a stale candidate s using LRU
4:         C ← {c : (s, c) ∈ S ∧ ℓ(c) is cached}
5:         if C ≠ ∅ ∧ Chance_s = 0 then
6:             Chance_s ← 1
7:             Pick another stale candidate s using LRU
8:         end if
9:         Evict ℓ(s) out from B
10:    end while
11: end procedure
```

5. PERFORMANCE EVALUATION

In this section, we present evaluation results for QTCA, and compare it with three baselines (see Section 2). All the experiments are run on a server with an Intel i7 930 at 2.8GHz, 6GB of memory and a 500GB disk (Seagate Barracuda, 7200rpm). Note that our experiments bypass the operating system buffering. Performance is measured in terms of the *cache hit ratio* as well as the *query response time*.

We use two thirds of the query log described in Section 3.1 as the training data to mine the co-occurrence of term pairs and carry out experiments on the remainder of query log (the involved posting lists total to around 3.67 GB). Further, we make use of 5 million web documents extracted from the *GOV2* collection as our document corpus, which results in an index of approximately 4.91 GB without positional information and frequencies. The SvS algorithm [7] is employed as the posting list intersection algorithm to evaluate the queries. Unless otherwise specified, experiments are conducted under the empirical setting[1]: $Sup = 1000$, $Conf = 0.5$, $Win = 1$.

[1]The experimental results under different *Sup* and *Conf* settings are omitted due to the space constraint.

Figure 2 depicts the average query response time of the four algorithms, while Figure 3 illustrates the cache hit ratio of different algorithms. We can see that QTCA results in an slight improvement in cache hit ratio when compared to other methods. QTCA thus offers the trade-off: at the cost of extra computation, QTCA can utilize term pair co-occurrence data. As indicated by the experimental results, by utilizing this extra information we can achieve a higher hit ratio. Moreover, the cost of extra computation is not particularly large. For example, it only takes several seconds to process the 1 million training queries using Algorithm 1.

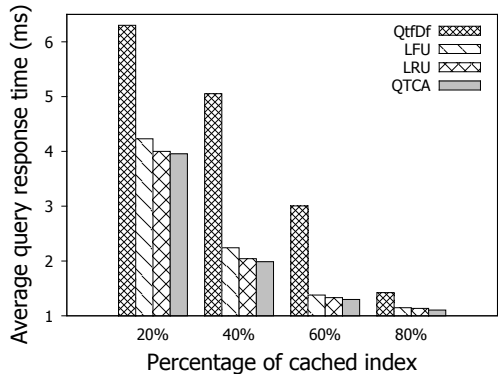

Figure 2: Average query response time of QTCA compared with other existing caching strategies.

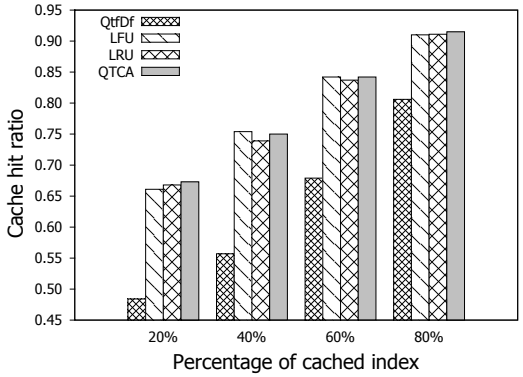

Figure 3: Cache hit rate of QTCA compared with other existing caching strategies.

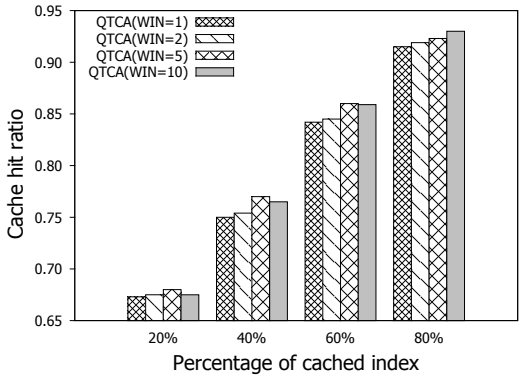

Figure 4: Cache hit rate of intra-query vs. inter-query QTCA (with varying WIN)

We also investigate the performance of QTCA under various window size settings. We consider the cases of $Win = 1$ ($QTCA$-intra) and $Win \in \{2, 5, 10\}$ ($QTCA$-inter). Figure 4 plots the performance of QTCA under these conditions. We see that there is little difference between the performance of QTCA-intra and QTCA-inter with $Win = 2$, regardless of the cache size. However, for larger cache sizes, the performance of QTCA-inter is better with a larger window size.

6. CONCLUSIONS AND FUTURE WORK

Motivated by the observation that the frequency of term pairs in query logs follows an approximate power law distribution, we present a caching strategy, QTCA, which aims to take advantage of this property. Experimental results show that QTCA is superior to any existing caching algorithms in terms of both cache hit ratio and query response time.

In this work, we focus on the co-occurring pairs of terms. The co-occurrence of three or more terms is beyond the scope of this paper, and is left as a future work. Another interesting open problem is to investigate the impact of query log's freshness on QTCA strategy.

7. ACKNOWLEDGMENTS

We would like to thank Xin Li for the initial version of correlation mining program and Eric Lo for the valuable feedback. Stones would also like to thank AARMS. This work is partially supported by NSFC of China (60903028, 61070014) and Key Projects in the Tianjin Science & Technology Pillar Program (11ZCKFGX01100).

8. REFERENCES

[1] R. A. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. The impact of caching on search engines. In *SIGIR*, 2007.

[2] R. A. Baeza-Yates and S. Jonassen. Modeling static caching in web search engines. In *ECIR*, 2012.

[3] R. A. Baeza-Yates, F. Junqueira, V. Plachouras, and H. F. Witschel. Admission policies for caches of search engine results. In *SPIRE*, 2007.

[4] R. A. Baeza-Yates and F. Saint-Jean. A three level search engine index based in query log distribution. In *SPIRE*, 2003.

[5] M. Chau, Y. Lu, X. Fang, and C. C. Yang. Characteristics of character usage in chinese web searching. *Inf. Process. Manage.*, 45(1):115–130, 2009.

[6] S. Chaudhuri, K. W. Church, A. C. König, and L. Sui. Heavy-tailed distributions and multi-keyword queries. In *SIGIR*, 2007.

[7] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Experiments on adaptive set intersections for text retrieval systems. In *ALENEX*, 2001.

[8] Q. Gan and T. Suel. Improved techniques for result caching in web search engines. In *WWW*, 2009.

[9] X. Long and T. Suel. Three-level caching for efficient query processing in large web search engines. In *WWW*, 2005.

[10] E. P. Markatos. On caching search engine query results. *Computer Communications*, 24(2):137–143, 2001.

[11] R. Ozcan, I. S. Altingovde, B. B. Cambazoglu, F. P. Junqueira, and Ö. Ulusoy. A five-level static cache architecture for web search engine. *Inf. Process. Manage.*, 48(5):828–840, 2012.

[12] P. C. Saraiva, E. S. de Moura, R. C. Fonseca, W. M. Jr., B. A. Ribeiro-Neto, and N. Ziviani. Rank-preserving two-level caching for scalable search engines. In *SIGIR*, 2001.

[13] C. Silverstein, M. R. Henzinger, H. Marais, and M. Moricz. Analysis of a very large web search engine query log. *SIGIR Forum*, 33(1):6–12, 1999.

[14] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *WWW*, 2008.