The Impact of using Combinatorial Optimisation for Static Caching of Posting Lists

Casper Petersen, Jakob Grue Simonsen, and Christina Lioma

University of Copenhagen, Copenhagen, Denmark,
cazz@di.ku.dk, simonsen@di.ku.dk, c.lioma@di.ku.dk

Abstract. Caching posting lists can reduce the amount of disk I/O required to evaluate a query. Current methods use optimisation procedures for maximising the cache hit ratio. A recent method selects posting lists for *static* caching in a greedy manner and obtains higher hit rates than standard cache eviction policies such as LRU and LFU. However, a greedy method does not formally guarantee an optimal solution. We investigate whether the use of methods guaranteed, in theory, to find an approximately optimal solution would yield higher hit rates. Thus, we cast the selection of posting lists for caching as an integer linear programming problem and perform a series of experiments using heuristics from combinatorial optimisation (CCO) to find optimal solutions. Using simulated query logs we find that CCO yields comparable results to a greedy baseline using cache sizes between 200 and 1000 MB, with modest improvements for queries of length two to three.

Keywords: posting list, caching, combinatorial optimisation

1 Introduction

A *posting list* consists of a term t and $n \geq 1$ postings, each containing the ID of a document where t occurs, and other information required by the search engine's scoring function, e.g. the frequency of t in each document [6]. Posting list caching can reduce the amount of disk I/O involved [13, 14] in query processing, affords higher cache utilisation and hit rates than result caching [12], and can combine terms to answer incoming queries.

Our contribution: We show that static caching of posting lists can be modelled in a principled manner using constrained combinatorial optimisation (CCO), a standard method that has yielded great improvements in many fields [9, Chap. 35], and we provide a principled investigation of whether CCO would yield better solutions (preferably using modest extra computational resources) than greedy methods. Using simulated query logs for a range of cache sizes, we perform a sequence of experiments that show that results using combinatorial optimisation is comparable to the greedy baseline of Baeza-Yates et al. using 200-1000 MB cache sizes, with some modest improvements for queries of length two to three.

2 Related Work

Much prior work has been devoted to caching posting lists [3, 4, 5, 6, 10, 11, 13, 14, 15]. Zhang et al. [15] benchmark five posting list caching policies and find LFU (least frequently used – cache members are evicted based on their infrequency of access) to be superior, and that cache hit rates for static posting list are similar to the LFU, but with less computational overhead. An integrated cache that merges posting lists of frequently co-occurring terms to build new posting lists in the inverted index is used by Tolosa et al. [14]. Using a cost function that combines disk lookup and CPU time, the integrated cache improves performance over standard posting list caching by up to 40%. Combinatorial optimisation for caching has not been investigated to the same degree: Baeza-Yates et al. [5] cache query terms based on their frequencies in a query log, and obtain ≈20% reduction in memory usage without increasing query answer time. Baeza-Yates et al. [3] extend this approach by caching query terms using (i) their frequency in a query log weighted by (ii) their frequency in a collection. Posting lists with the highest weight are then cached. This method obtains higher hit rates than their approach in [5], dynamic LRU (least recently used) and dynamic LFU for all cache sizes. We propose an extension of [3] which uses a principled method to select posting lists for static caching. Next, we describe the original method by Baeza-Yates et al., and our extension.

3 Posting Lists Caching

Greedy Posting Lists Caching Consider a list of queries, each of which consists of one or more terms and a cache of finite capacity. Let $F_q(t)$ denote the number of queries that contain term t in some query log Q_L and $F_d(t)$ the number of documents that contain t in some collection C. A *greedy* strategy to posting list selection chooses the query terms (representing posting lists) with the highest $F_q(t)$ until cache space is exhausted as in [5]. However, Baeza-Yates et al. [3] observe a trade-off between terms with high $F_q(t)$ and high $F_d(t)$ as these have long posting lists that consume substantial cache space. They address this trade-off by using the ratio $F_q(t)/F_d(t)$, called QTFDF, to select terms for static caching by (i) calculating QTFDF of each $t \in Q_L \cap C$, (ii) sorting terms in decreasing value of QTFDF and (iii) caching the terms with the highest QTFDF until cache space is exhausted. The method of [3] is thus a clever variation of the *profit-to-weight ratio* approach first used by Dantzig [8].

Selecting which posting lists to load into the cache is a *0-1 knapsack problem* [3, 4]: given a knapsack with capacity c and n items $c_1, \ldots, c_n$ having values $v_1, \ldots, v_n$ and weights $w_1, \ldots, w_n$, take the items that maximise the total value without exceeding c. An item can only be selected once and fractions of items cannot be taken. As the knapsack optimisation problem is NP-hard and cannot in general be solved optimally using a greedy strategy [7, Chap. 16], we next describe how to formulate posting list selection as a combinatorial optimisation problem which, in theory, would find an approximately optimal solution.

Combinatorial Optimisation for Posting Lists Caching We formalise the observation of [3] that a trade-off exists between $F_q(t)$ and $F_d(t)$ as follows: terms should be cached that yield the highest possible $F_q(t)$ subject to the constraint that the total size of the posting lists of cached terms should not exceed cache size. This is a classic CCO problem (a fact already noted by [3], but without formalisation or reported experiments). We cast posting list selection as an *integer linear program* of the form:

$$\max \quad \sum_{i=1}^{n} v_i x_i \tag{1}$$

$$\text{subject to} \quad \sum_{i=1}^{n} w_i x_i \leq c \tag{2}$$

$$x_i \in \{0,1\}, 1 \leq i \leq n \tag{3}$$

where $\sum_{i=1}^{n} v_i x_i$ is the *objective function*, $\sum_{i=1}^{n} w_i x_i \leq c$ and $x_i \in \{0,1\}, 1 \leq i \leq n$ are *constraints* where x_i represents a term t_i (a posting list). A *solution* is a setting of the variables x_i; a *feasible* solution is a solution that satisfies all constraints; and an *optimal* solution is a feasible solution with maximal value of the objective function. We consider only optimal solutions here. Eq. (2) states that the total weight of the selected terms cannot exceed c, and Eq. (3) that each term is either selected or discarded. We set $v_i = F_q(t_i)$ and $w_i = F_d(t_i)$, and refer to the method described here as the *CCO* method.

We emphasise two points. First, the CCO method maximises the chance of a query term cache hit, but does not consider disk I/O a factor. We can do this using a *multi*-objective CCO problem where one objective function seeks to minimize disk I/O (using the length of the posting list of x_i as values v_i) and a second objective function that seeks to maximise the number of cache hits. Second, if a term is selected its *entire* posting list is loaded. Another approach is to allow fractions of posting lists to be loaded and access the main index as needed. This may be useful if e.g. each posting list is sorted so access to the main index is reduced. We leave both topics as future work.

4 Simulating queries

Query logs from large search engines are typically not publicly available in large numbers. Instead, we construct simulated query logs using (i) the method of Azzopardi et al. [2] and (ii) random sampling from a large synthetic query log.

Known-item queries We construct synthetic query logs containing known-item queries using the method of [1, 2] as follows: We first select a document d_k from the collection (with uniform probability), then select a query length l and then select l terms $t_{1,\ldots,l}$ from the document language model (LM) of d_k with probability $p(t_i|\Theta_d)$ and add t_i to q. $p(t_i|\Theta_d)$ is a mixture of (i) the maximum likelihood estimate of a term occurring in a document and (ii) a background model $p(t)$ (maximum likelihood estimate of t in the collection). Estimating (i) is done using one of two LMs [1]. The *popular* LM is given by

$$p(t_i|d_k) = n(t_i, d_k) / \sum_{t_j \in d_k} n(t_j, d_k) \tag{4}$$

where t_i, t_j are terms in d_k and $n(t_i, d_k)$ is the term-frequency of t_i in d_k. The *discriminative* LM is given by

$$p(t_i|d_k) = b(t_j, d_k)/p(t_i) \cdot \sum_{t_j \in d_k} b(t_j, d_k)/p(t_j) \quad (5)$$

where $b(t_j, d_k) = 1$ if term t_j occurs in d_k.

Sampling from a large query log We use the anchor text query log from ClueWeb09[1] as starting point, which contains 500M triplets of the form <`URL`, `anchor text`, `fq`> where `fq` is the frequency of the tuple <`URL`, `anchor text`>. From this query log, we sample with replacement to generate new query logs.

5 Experiments

We describe how we simulate repeated queries and how we measure performance. We evaluate the CCO method against the greedy baseline of Baeza-Yates et al. [3], using the number of cache hits as our cache performance measure.

Simulating repeated queries The method of Section 4 generates queries occurring exactly once. To generate *repeated* queries in the synthetic query sets we do as follows: after simulating a query, we generate a random number r in the interval $(0; 1)$ and compare it to a threshold τ. If $r > \tau$ we duplicate the query. We fix $\tau = 0.44$ meaning that $\sim 56\%$ of the queries have multiple occurrences [4]. We simulate queries of length $l = 1, 2, 3$ and generate $m = 5$ queries from each document. For query logs simulated using the method in Section 4, we cannot control repeated queries.

Experimental settings We experiment with cache sizes of 200, 600 and 1000 MB (cache sizes can vary between 100 MB to 16 GB [14]) and fix the size of a posting to 8 bytes. We use ClueWeb09 cat. B. – a domain-free crawl of ca. 50 million web pages in English – indexed using INDRI 5.8 with no stemming and with stop words removed as collection. We simulate query logs of 1M, 5M and 10M queries using each of the two LMs from Section 4 and the method from Section 4 with the anchor text as queries. As in [3], we estimate $F_q(t)$ from each query log and $F_d(t)$ from the collection. Each CCO problem is solved using SYMPHONY[2] (extensive experiments and tuning using LP_SOLVE[3] gave no consistent improvements). We count a cache hit for a query iff *at least one* of its terms is found in the cache (see [15] for alternative definitions). A single hit is sufficient for efficient retrieval as we need only traverse that term's posting list, and scan the forward index of each document to determine if remaining query terms are found. Counting query hits using this linear scan approach is less efficient than posting list intersection, but in this preliminary work, it allows us to test the merit of our method.

[1] `http://lemurproject.org/clueweb09/anchortext-querylog/`

[2] `https://projects.coin-or.org/SYMPHONY`

[3] `http://lpsolve.sourceforge.net/5.5/`

6 Findings

We show results for the 5M and 10M query logs generated using the method from Section 4 in Table 1. Results for all other query logs are qualitatively similar. We do not report CPU or memory consumption as this cost is likely minimal compared to indexing and retrieval costs. Across all query logs, query lengths (qlen) and cache sizes, the overlap coefficient is $> 85\%$ and both CCO and the baseline cache contain approximately the same number of terms. For qlen=1, CCO and the baseline perform nearly identically for all query logs. For qlen=2, the discriminative query log gives rise to the largest differences between CCO and the baseline though these differences are negligible relatively to the total number of cache hits. For the popular query log, the differences are substantially smaller. The observations for qlen=3 are identical to those for qlen=2.

		Simulated 5M					
		Discriminative			**Popular**		
		qlen=1	qlen=2	qlen=3	qlen=1	qlen=2	qlen=3
OC	200M	0.851	0.884	0.923	0.990	0.973	**0.977**
	600M	0.962	**0.934**	0.899	0.997	**0.987**	**0.999**
	1000M	0.952	**0.952**	**0.942**	**0.995**	**0.994**	**0.998**
CT	200M	24938/24951	24922/24920	24973/24974	9311/9310	13799/13782	**16841/16845**
	600M	74648/74483	**74725/74740**	74780/74752	13804/13803	**21483/21473**	**27568/27557**
	1000M	114269/115850	**122655/123358**	**124525/124546**	**16309/16307**	**25704/25696**	**33194/33209**
CH	200M	217626/217626	228183/228266	228692/228877	19185/19185	28655/28671	**35242/35238**
	600M	546566/546566	**596812/596790**	600376/600733	28537/28537	**44579/44573**	**57352/57351**
	1000M	765793/765793	**859142/857718**	**880740/880132**	**33768/33767**	**53531/53529**	**69370/69366**
DIFF	200M	0	-83	-185	0	-16	**4**
	600M	0	**22**	-357	0	**6**	**1**
	1000M	0	**1424**	**608**	**1**	**2**	**4**
		Simulated 10M					
		Discriminative			**Popular**		
		qlen=1	qlen=2	qlen=3	qlen=1	qlen=2	qlen=3
OC	200M	0.949	0.957	0.865	0.961	**0.929**	0.977
	600M	0.890	**0.909**	**0.885**	0.989	0.982	**0.985**
	1000M	**0.910**	**0.891**	0.957	**0.999**	**0.989**	**0.999**
CT	200M	24988/24958	24955/24926	24954/24935	13892/13886	**19404/19335**	22775/22799
	600M	74589/74537	**74468/74591**	**74642/74562**	21623/21614	32962/32919	**41095/41170**
	1000M	**124251/124127**	**123925/124091**	124632/124350	**25872/25876**	**40122/40096**	**51247/51247**
CH	200M	240056/240056	246439/246440	246117/246293	28634/28634	**40152/40145**	48251/48258
	600M	629405/629405	**662230/662148**	**664890/664515**	44710/44710	68199/68203	**86191/86154**
	1000M	**957964/957963**	**1033190/1033109**	1044362/1044852	**53631/53629**	**83379/83374**	**107333/107332**
DIFF	200M	0	-1	-176	0	**7**	-7
	600M	0	**82**	**375**	0	-4	**37**
	1000M	**1**	**81**	-490	**2**	**5**	**1**

Table 1: Results for the Discriminative and Popular 5M and 10M query log for query lengths = 1,2,3 and cache sizes: 200,600 and 1000 MB. *x/y* means *CCO / baseline*. OC is the *overlap coefficient* $= \frac{|X \cap Y|}{\min(|X|,|Y|)}$. CT is the number of *cache terms*. CH is the number of *cache hits*. *Diff* is the difference in CH. Entries where *Diff*>0 (boldfaced).

7 Conclusions and Future Work

We have investigated static posting list caching as a constrained combinatorial optimisation (CCO) problem and have evaluated this theoretically principled method against the greedy method of Baeza-Yates et al. [3]. We found both

methods performed similarly for all cache sizes, with some modest gains for the CCO method. The high values (>85%) of the overlap coefficient in all experiments suggest that both methods mostly identify the *same* high-frequency query terms and that differences in cache hits can be attributed to a small set of infrequent terms. However, while combinatorial optimisation gives, in theory, optimal solutions, in practice the quality of the solution also depends on the problem, the solver and the settings of the solver's parameters. In future work, we will investigate (i) how this impacts posting list selection, (ii) if CCO can obtain consistent performance improvements for domain-specific query logs, and (iii) the use of multi-objective CCO to balance disk I/O with cache hits and allowing fractions of posting lists to be cached.

References

[1] Leif Azzopardi and Maarten de Rijke. Automatic construction of known-item finding test beds. In *SIGIR*, pages 603–604, 2006.

[2] Leif Azzopardi, Maarten de Rijke, and Krisztian Balog. Building simulated queries for known-item topics: an analysis using six european languages. In *SIGIR*, pages 455–462, 2007.

[3] Ricardo Baeza-Yates, Aristides Gionis, Flavio Junqueira, Vanessa Murdock, Vassilis Plachouras, and Fabrizio Silvestri. The impact of caching on search engines. In *SIGIR*, pages 183–190. ACM, 2007.

[4] Ricardo Baeza-Yates, Aristides Gionis, Flavio P. Junqueira, Vanessa Murdock, Vassilis Plachouras, and Fabrizio Silvestri. Design trade-offs for search engine caching. *TWEB*, 2(4):20, 2008.

[5] Ricardo Baeza-Yates and Felipe Saint-Jean. A three level search engine index based in query log distribution. In *SPIRE*, pages 56–65. Springer, 2003.

[6] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. Efficient query evaluation using a two-level retrieval process. In *IKM*, pages 426–434. ACM, 2003.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT Press Cambridge, 2001.

[8] George B. Dantzig. Discrete-variable extremum problems. *Operations research*, 5(2):266–288, 1957.

[9] Martin Grotschel and László Lovász. Combinatorial optimization. *Handbook of combinatorics*, 2:1541–1597, 1995.

[10] Zhen Liu, Philippe Nain, Nicolas Niclausse, and Don Towsley. Static caching of web servers. In *PWEI*, pages 179–190. ISOP, 1997.

[11] Xiaohui Long and Torsten Suel. Three-level caching for efficient query processing in large web search engines. *WWW*, 9(4):369–395, 2006.

[12] Myron Papadakis and Yannis Tzitzikas. Answering keyword queries through cached subqueries in best match retrieval models. *JIIS*, pages 1–40, 2014.

[13] Paricia C. Saraiva, Edleno Silva de Moura, Novio Ziviani, Wagner Meira, Rodrigo Fonseca, and Berthier Riberio-Neto. Rank-preserving two-level caching for scalable search engines. In *SIGIR*, pages 51–58. ACM, 2001.

[14] Gabriel Tolosa, Luca Becchetti, Esteban Feuerstein, and Alberto Marchetti-Spaccamela. Performance improvements for search systems using an integrated cache of lists+intersections. In *SPIRE*, pages 227–235. 2014.

[15] Jiangong Zhang, Xiaohui Long, and Torsten Suel. Performance of compressed inverted list caching in search engines. In *WWW*, pages 387–396. ACM, 2008.