Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

C3C: A New Static Content-Based Three-Level Web Cache

THANH TRINH, DINGMING WU, AND JOSHUA ZHEXUE HUANG
College of Computer Science and Software Engineering, Shenzhen University, China
Corresponding author: DingMing Wu (e-mail: dingming@szu.edu.cn).

ABSTRACT One way of improving the performance of a search engine is increasing the hit ratio of the search engine cache. A common and widely used approach for increasing the hit ratio is a combination of the result cache, the posting list cache and the intersection cache, yielding a three-level cache architecture. However, the existing multi-level cache architectures do not consider the dependencies among the content cached in different parts. Thus, the same content might be stored multiple times in the architectures, resulting in duplicate hits. In other words, a large amount of space in the cache is wasted. In this paper, we propose a new static content-based three-level cache architecture that adopts a new C3C filling algorithm that takes into account the dependencies among the content cached in different parts. In the proposed cache architecture, duplicate hits are reduced and the hit ratio increases. Extensive experiments were conducted on a real data set. The results have shown a significant improvement on the hit ratios compared with two existing approaches.

INDEX TERMS Cache Design, Intersection Cache, List Cache, Result Cache

I. INTRODUCTION

Search engines are used as a prevalent way to find information on the Internet by submitting a query to the engines. Nowadays, search engines process tremendous queries hourly. The volume of queries increases steadily over time, so how to process a large number of queries over a huge amount of data in a short execution time is a real challenge. To enhance the performance of search engines, caching techniques are frequently used.

A number of caching techniques used in search engines have been studied so far. Current search engines often use five popular caching techniques, namely result cache [1]–[5], posting list cache [6], [7], intersection cache [8], [9], snippet cache [10], and document cache [11]. The well-known cache replacement strategies include LRU, LFU and LCU [12]. Regarding the cache filling strategies, well-known ones include Freq-Based [1], Freq/Size [6], Cost-based, Freq×Cost/Size and Freq×Cost$^{2.5}$. In order to achieve further performance gains, different multi-level cache architectures are proposed, for example the two-level, the three-level, as well as the five-level architectures [13]–[19], which are combinations of several caching techniques. However, the caching mechanisms of these cache architectures often ignore the relationships of the cached contents in the separate caches. For instance, the result cache stores the results of a query with some terms that are also stored in the posting list cache. In this case, we have duplicates in both caches and the cache space is wasted.

The cache architectures that adopt different static caching mechanisms affect the performance of query processing in different ways. Currently, there are two kinds of caching mechanisms used to fill up the content of separate caches. One caching mechanism fills the contents of separate caches independently without considering the relations of the contents [6], [8], [13]–[15], [20]. The order of filling separate caches is not considered in this caching mechanism. Another caching mechanism was proposed in the five-level cache architecture [17]. In this caching mechanism, the relations of the contents in separate caches are evaluated by the scores of the cached items. The score of an item is calculated from the frequency of the item, the time to fetch the item from the inverted index, and the cache space needed to store the item. Since an item is chosen for storing, the scores of all remaining items that are affected are recalculated. Although this mechanism improves query latency, it does not reduce duplicate hits.

In this paper, we propose a new static three-level cache architecture that uses a new content-based caching mechanism to make best use of the cache space and remove duplicate hits. The proposed web cache consists of three parts, i.e., the result cache (RC), the posting list cache (PC), and the

intersection cache (IC). In the proposed cache architecture, we propose three cache filling algorithms to fill RC and PC to reduce duplicate hits. In addition, we propose a new term set generation algorithm (TSG) that generates sets of terms from the queries in the query log. The generated term sets are sorted in the descending order of their scores, where the score of each term set is calculated based on the frequency of the term set and the size of the cache space needed to store the intersection of the posting lists of the terms in the set. The term sets with high scores are cached in IC. We propose four ways of filling IC that choose different lengths of the term sets.

Experiments were conducted on a real AOL query log and the document collections of Wikipedia [1]. The cache sizes of $200MB$ and $500MB$ were used in the experiments. The experimental results have shown that the proposed web architecture has improved the hit ratio by 7% over the two existing methods.

The remainder of this paper is organized as follows. Section II reviews the related work. Section III presents the architecture of the proposed web cache. Section IV proposes three algorithms for establishing the dependency between RC and PC, the term set generation algorithm that provides candidates for the intersection cache (IC), and the C3C filling algorithm that establishes the dependencies among RC, PC, and IC. Experiment results are included in Section V. Section VI provides conclusions of this work.

II. RELATED WORK

Result Cache. It is used to cache the results of previously submitted queries. When a new query is issued to the search engine and if the results of this query are cached, the search engine will return these results immediately. Markatos [1] studied queries from EXCITE search engine and found that there are a considerable number of these queries with similar terms. The study also shows that the results of queries with high frequency could be stored in medium cache sizes, and a small cache is effectively for static caching techniques. Static dynamic cache that contains static cache and dynamic cache parts was proposed by Fagni et al. [2], in which the only-read static part caches the submitted queries with high frequency and the other part is used for a chosen replacement strategy. Altingovde et al. [3] proposed a cost-aware caching strategy for the static caching based on results of queries and their execution time. Gan and Suel [4] studied the weighted problem of caching results in order to maximize cost savings instead of hit ratios, and proposed a feature-based method that provides improved results. Ozcan et al. [5] studied a query frequency problem to maximize the hit ratio for the result cache, then provided a more accurate feature established based on the stability of query frequency over several given time intervals. Kucukyilmaz et al. [21] studied the feature extraction problem from Yahoo query log and proposed a machine learning model to improve the hit ratio of the result cache.

Posting List Cache. This cache is another way to retrieve the results of a submitted query if not found in the result cache. The posting list cache is used to store the lists of terms of submitted queries. Since a new submitting query is formed by several cached terms, the cache returns result identifiers of the query intermediately. Baeza-Yates et al. [6], [20] studied the trade-off problem, and found that storing posting lists of terms would achieve a higher hit ratio compared to cache query results in medium-size caches. They also proposed a new static caching method to store posting lists which yields improved results. Zhang et al. [7] studied inverted index compression and index caching problems. Machine parameters that affect the performance of combing the two techniques are discovered, such as CPU speed and disk speed.

Intersection Cache. The intersection cache (IC) works similar to the posting list cache. However, IC stores the intersections of the posting lists for some term sets formed by several terms. If a submitting query is made by the cached term sets, IC will return result identifiers of the query. Therefore, the execution time of the query is decreased. Search engines integrated with IC can improve their performance. Zhou et al. [8] explored the item sets that appear frequently in queries, and suggested a new replacement policy for the intersection cache. A three-level cache that contains the proposed intersection cache was proposed, and improved results were obtained. Tolosa et al. [9] studied the integrated cache that combines the posting list cache and the intersection cache in a single stored space. The term pairs are structured to make an efficient use of the cached space in order to maximize the hit ratio.

Snippet Cache. Document servers often use two types of caches in order to return queries results, i.e., Snippet cache and Document cache [22]. In general, each result of a query contains a summary related to the search keywords, i.e., a snippet. The snippets that have been produced from previous queries are stored in the snippet cache. Ceccarelli et al. [10] defined a concept of supersippets, which were formed by several sentences in documents, and were presented to answer coming queries.

Document Cache. This cache is used in documents servers to store several previously retrieved documents. Next time when the document is requested again, the time of retrieving it from the disk is saved. Ozcan et al. [11] proposed a hybrid dynamic result cache consisting of two sections: a docID cache and an HTML cache.

Multiple Level Cache. To take the advantages of storing results and storing posting lists, Saraiva et al. [13] designed a two-level caching architecture that was formed by these two caches. Baeza-Yates and Jonassen [14] suggested an optimal way to split a given stored space for the result cache and the list cache. Dong et al. [18] proposed new data selection for caching results and posting lists, and designed a two-level cache architecture. SSD was used as a secondary memory in this architecture in order to improve the performance

[1] https://dumps.wikimedia.org/enwiki/20161201

of search engines. Detti et al. [19] studied the impact of invalidation mechanisms on the LRU caches and proposed the hit probability model based on Poisson distribution. Then this model was used in the two-level hierarchical cache. In the work of Long and Suel [15], the high frequent term sets appeared in queries are first exploited to be stored in the intersection cache, and a three-level cache was proposed to combine the result cache, the posting list cache and the static intersection cache. Marín et al. [16] designed a cache hierarchy architecture to efficiently process user queries. The different caches in this architecture store diverse cached items which address frequent queries effectively, such as segments of index and query answers. Ozcan et al. [17] proposed a multi-level static cache architecture that combines five different caches, i.e., the result cache, the posting list cache, the scored cache, the intersection cache and the document cache. They then provided an optimal way to split a given memory for the five caches.

Cache Replacement Strategy. Based on an overview of different cache replacement proposals, Podlipnig and Böszörményi [12] defined five main groups, namely frequency-based (e.g., LFU), recency-based (e.g., LRU), recency/frequency-based (e.g., LRU*), randomized strategies and function-based. Ma et al. [23] proposed a new weighting size and cost replacement policy (WSCRP), which was an extension of LFU. Hasslinger et al. [24] proposed a class of score gate least recently used (SG-LRU) for web caching strategies. The hit ratios achieved by SG-LRU were better than that of LRU. Bechmann et al. [25] studied the hit density problem and proposed a new cache replacement, namely Least Hit Density.

Cache Filling Strategy. Each filling strategy uses a different algorithm to compute a score for each item. The items with high scores are selected to fill up the caches. The five following popular strategies are listed .

- FB (Freq-Based): This strategy adds popular items, such as queries or terms, to corresponding caches.
- FS (Freq/Size): Each item is assigned a score by a fraction of its frequency and its size, then the items with high scores are selected to fill up the caches [6].
- Cost (Computation Time): To reduce the execution times of items (queries or terms) to load results from servers, the items with high computed time are cached in this strategy [26].
- FC ($FB*Cost^{2.5}$): It computes score $FB*Cost^{2.5}$ for each item, then the items with high scores are put into the caches.
- FCS (Freq*Cost/Size): It calculates score Freq*Cost/Size for each item, then the items with high scores are stored in the caches.

III. ARCHITECTURE OF CONTENT-BASED THREE-LEVEL WEB CACHE

The Content-based three-level web Cache (C3C) architecture consists of three components (illustrated in Fig. 1), i.e, a result cache (RC), an intersection cache (IC), and a posting list cache (PC). The RC stores the results of some queries previously submitted, where for a query, each of its results includes the title, URL, and the snippet of the corresponding web page. The PC stores the posting lists $L(t_i)$ of some terms t_i that appear in submitted queries. The intersection cache stores the IDs of the intermediate results of some previous queries. The intermediate results refer to the list of IDs of the web pages that contain a set of terms T that appear in previous queries. There exist dependencies among the content cached in these three components. The methods to establish the dependencies is presented in Section IV. The C3C is the main memory resident. When a query Q arrives, it works in the following way:

Step 1: **Result Cache Checking**: If the query is stored in the result cache, The query's results are returned to the client; otherwise, go to the next step.

Step 2: **Intersection Cache Checking**: If a set of term sets $\mathbf{T} = \{T_i\}$ can be found in the intersection cache, such that $\bigcup_{T_i \in \mathbf{T}} T_i = Q$, the result web pages are determined using $\bigcap_{T_i \in \mathbf{T}} L(T_i)$ and returned to the client; otherwise, let Q_m be the set of missed terms, Q_h be the set of hit terms in the intersection cache, and $L(Q_h)$ be the list of web page IDs that contain Q_h. Go to the next step.

Step 3: **Posting List Cache Checking**: If all the terms in Q_m can be found in the posting list cache, the result web pages are determined using $\bigcap_{t_i \in Q_m} L(t_i) \bigcap L(Q_h)$ and returned to the client; otherwise, the missed terms will be passed to the next step.

Step 4: **Inverted Index Reading**: Fetch the posting lists of the missed terms from the inverted index.

IV. DEPENDENCIES ESTABLISHMENT IN C3C

This section presents the algorithms for establishing the dependencies among the three components in the C3C.

A. DEPENDENCY BETWEEN RC AND PC

In this section, we propose three cache filling algorithms that establish the dependency between the result cache and the posting list cache. These algorithms differ in terms of the ways of considering the dependencies.

1) Query Oriented Filling (QOF)

Algorithm QOF aims at reducing the response time of frequent queries with small size of results. It chooses the cache where less space is used. For instance, given a query q, if the space used to store the results of q is less than the space used to store the posting lists of all terms in q, the RC will be chosen to store the results. Otherwise, the posing lists of all terms are put into the PC. In addition, the content of RC depends on the content of PC, which means that if PC has stored all the terms of a query, the algorithm will not store its results in RC any more. Algorithm 1 describes the pseudo code of algorithm QOF. It consists of the following three steps.

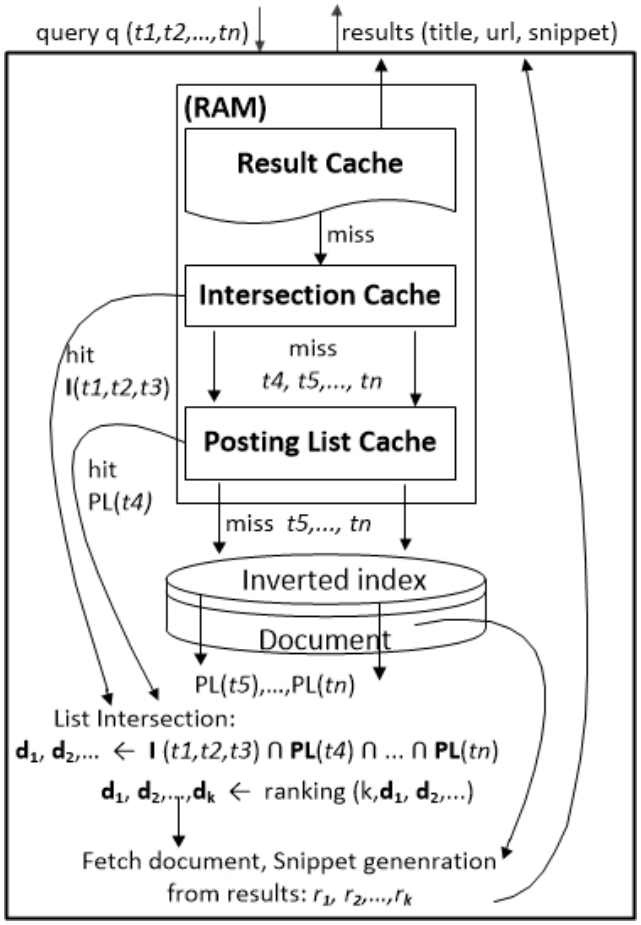


FIGURE 1: Content-based Cache Architecture.

Step 1: **Scoring:** For each distinct query q and each distinct term t, a score is computed as $score = freq/size$, where $freq$ is the frequency of query q or term t in the query log and $size$ is the cache space needed for storing the results of query q in RC or the cache space needed for storing the posting list of term t in PC. All the queries and all the terms are sorted in the descending order of their scores, respectively.

Step 2: **Filling RC and PC:** While the RC is not full, algorithm QOF processes the queries in the descending order of their scores. For the current query under processing, if all the terms in the query have been cached in the PC, this query is discarded. Otherwise, algorithm QOF calculates the space needed for caching the results of the query in RC, denoted by W and the space needed for caching the posting lists of the missed terms of the query in PC, denoted by T. The algorithm chooses the cache with less space needed. In other words, if $W \geq T$, the posting lists of the missed terms of the query will be cached in PC, otherwise, the results of the query will be cached in RC.

Step 3: **Filling PC:** This step is optional and is only needed when RC is full, but PC still has an amount of free space. The PC is filled up with the posting lists of terms with high scores.

Algorithm 1 QOF

Result cache $RC \leftarrow \emptyset$
Posting list cache $PC \leftarrow \emptyset$
Compute $score_Q \leftarrow freq_Q/size_Q$ for distinct query
Compute $score_T \leftarrow freq_T/size_T$ for distinct term
while RC *is not full* **do**
  $q \leftarrow \mathbf{getNextQuery}()$
  if PC *stored the posting lists of all terms in* q **then**
    continue
  end
  $T \leftarrow$ cache space used to store the posting list of missed terms in PC
  $W \leftarrow$ cache space used to store the query's results in RC
  if $W \geq T$ **then**
    PC stores the posting lists of missed terms
  else
    add the results of q to RC
  end
end
while PC *is not full* **do**
  $t \leftarrow \mathbf{getNextTerm}()$
  add the posting list of t to PC
end

Example 1. Tables 1 and 2 are used to illustrate how algorithm QOF fills RC and PC. The scores of the queries and terms are shown in the tables. Fig. 2 shows the cache filling process of algorithm QOF. The first query to be processed is q_1. In the beginning, the content of PC is blank, hence all the terms in q_1 are not stored. PC uses the size of $T = 60KB$ to cache the whole posting lists of these terms. To store the results of q_1, RC uses $W = 30KB$. Since $W < T$, RC is selected to store the results of q_1. The following query q_2 is now processed. Because PC uses less space than RC, PC is used to store the posting lists of the terms in q_2. The next query to be processed is q_3. Since PC stored all the terms in q_3, it does not make sense to store q_3 and q_3 is discarded. Finally, for query q_4, PC does not store any term in q_4. More cache space is used in PC, compared with RC, i.e., $W = 20KB < T = 35KB$, RC is then chosen to store the results of q_4.

		RC	PC
q_1: (apple, banana,lemon)	W = 30KB < T = 60KB	q_1	
q_2: (toy, blue, orange)	W = 30KB > T = 15KB	q_1	toy, blue, orange
q_3: (orange, blue)	found in **PC**	q_1	toy, blue, orange
q_4: (green, car)	W = 20KB < T = 35KB	q_1 q_4	toy, blue, orange

FIGURE 2: Running Example of Algorithm QOF.

TABLE 1: Example Queries and Scores.

Queries	Frequency	Size (KB)	Score	Normalized Score
q_1: (apple, banana, lemon)	250	30	8.33	1
q_2: (toy, blue, orange)	249	30	8.3	0.98
q_3: (orange, blue)	200	30	6.67	0.8
q_4: (green, car)	120	20	6.0	0.7

TABLE 2: Example Terms and Scores.

Terms	Frequency	Size (KB)	Score	Normalized Score
t_1: blue	5000	2	2500	1
t_2: car	9500	10	950	0.9
t_3: orange	7200	8	900	0.85
t_4: mouse	8800	10	880	0.79
t_5: toy	4000	5	800	0.69
t_6: green	19500	25	780	0.67
t_7: apple	11400	15	760	0.63
t_8: banana	17500	25	700	0.59
t_9: lemon	13000	20	650	0.55
t_n: ...	...	...	...	0.0

2) Term Oriented Filling (TOF)

Algorithm TOF firstly stores the posting lists of the terms with high frequency and small storing space in PC. According to the literature, caching posting lists of frequent terms would achieve a higher hit ratio compared to caching the results of frequent queries. When PC is full, the algorithm continues to fill RC. In order to better utilize the space, when deciding whether or not storing the query's results in RC, the content of PC is referred. Similar to algorithm QOF, if PC has stored all the terms in the query, this one is discarded. The pseudo code of algorithm TOF is briefly described in Algorithm 2, which includes the following three steps.

Step 1: **Scoring:** The scores of terms and queries are calculated in the same way as the first step in algorithm QOF. All the terms and queries are sorted in the descending order of their scores, respectively.

Step 2: **Filling PC:** PC is filled up with the posting lists of terms with high scores.

Step 3: **Filling RC:** The queries are processed in the descending order of their scores. For each query under processing, the content of PC is examined to verify how many terms of the query have been cached. If all the terms in the query are found in PC, this query is discarded. Otherwise, the results of the query are added to RC.

Example 2. Tables 1 and 2 are still used to demonstrate the process of Algorithm TOF. The process is shown in Fig. 3. Firstly, PC is filled up with the posting lists of the four terms with high scores, i.e., 'blue', 'car', 'orange', and 'mouse'. Next, the queries are respectively processed. For q_1, its terms are missed in PC, RC is then selected to store its results. Similar to query q_2, thus the results of q_2 are put into RC.

Algorithm 2 TOF

```
Query result cache RC ← ∅
Posting list cache PC ← ∅
Compute score_Q ← freq_Q / size_Q for distinct query
Compute score_T ← freq_T / size_T for distinct term
while PC is not full do
    t ← getNextTerm()
    add the posting list of t to PC
end
while RC is not full do
    q ← getNextQuery()
    if PC stored the posting lists of all terms in q then
        continue
    else
        add the results of q to RC
    end
end
```

Because PC stores all the terms in q_3, q_3 is discarded. Due to the same reason as q_1 and q_2, RC stores the results of q_4.

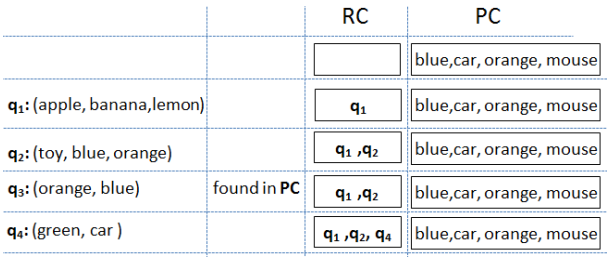


FIGURE 3: Running Example of Algorithm TOF.

3) Score Oriented Filling (SOF)

Different from algorithms QOF and TOF that sort queries and terms separately, algorithm SOF poses a unified order on both queries and terms according to their scores. All the queries and terms are put into a list in the descending order of their scores (normalized into range $[0, 1]$). In this algorithm, we use an entry to refer to either a query or a term in the sorted list. If the retrieved entry from the list is a term and the PC is still not full, PC will store the posting list of this term. If the retrieved entry from the list is a query, the algorithm first checks whether PC stores all the terms in the query. If yes, the query is discarded, otherwise, the space used to store it in both PC and RC is calculated. The cache with less space used is selected. If the selected cache is full, the other one is chosen. The algorithm stops when both RC and PC are full. Algorithm 3 describes the pseudo code of algorithm SOF. It includes the following two steps.

Algorithm 3 SOF

Query result cache $RC \leftarrow \emptyset$
Posting list cache $PC \leftarrow \emptyset$
Compute $score_Q \leftarrow freq_Q / size_Q$ for distinct query
Compute $score_T \leftarrow freq_T / size_T$ for distinct term
Normalize the scores of terms into range $[0, 1]$
Normalize the scores of queries into range $[0, 1]$
$L \leftarrow$ Add all the terms and queries to a sorted list in the descending order of their normalized scores
while *RC is not full $\vee$ PC is not full* **do**
  $e \leftarrow$ **getNextEntry**()
  if *e refers to a query* **then**
    if *not all the terms in e are found in PC* **then**
      if *PC is full* **then**
        add the results of query q to RC
        continue
      end
      if *RC is full* **then**
        add the posting lists of the terms in e to PC
        continue
      end
      $T \leftarrow$ space used to store the posting lists of missed terms in PC
      $W \leftarrow$ space used to store the query's results q in RC
      if $W \geq T$ **then**
        add the posting lists of the missed terms to PC
      else
        add the results of query q to RC
      end
    end
  end
  if *e refers to a term* **then**
    if *PC is not full* **then**
      add the posting list of e to PC
    end
  end
end

Step 1: **Scoring:** The scores of queries and terms are computed in the same way as those in algorithms QOF and TOF. The scores of each query and term are normalized into range $[0, 1]$. The queries and terms are sorted in the descending order of their normalized scores.

Step 2: **Filling:** Each time the algorithm removes the first entry from the sorted list containing queries and terms. If the entry refers to a term and PC is not full, the posting list of the term is added to PC. If the entry refers to a query and not all the terms in the entry are cached in PC, the algorithm will choose either PC or RC. If PC is full and RC is not full, the results of the query are added to RC. If RC is full and PC is not full, the posting lists of the terms in the query are added to PC. If both PC and RC are not full, the cache with less space needed is chosen. The algorithm terminates when both PC and RC are full.

Example 3. Tables 1 and 2 are used to explain the process of Algorithm SOF. Fig. 4 shows how PC and RC are filled by using algorithm SOF. The queries and terms are put into one list in the descending order of their normalized scores. The algorithm processes entries (queries and terms) in the sorted list one by one. If multiple entries have the same normalized score, we just arbitrarily decide their order. In this example, the first entry under consideration is term 'blue'. The posting list of 'blue' is put into PC. The next entry removed from the list is query q_1. Since all the terms in q_1 are not stored in PC, RC stores the results of q_1. Similarly, RC stores the results of q_2 and the posting lists of 'car' and 'orange' are stored in PC. When q_3 is removed from the list, the algorithm finds that all the terms in q_3 have been existed in PC, so that q_3 is discarded. Finally, the posting list of 'mouse' is put into PC.

		RC	PC
t_1: blue			blue
q_1: (apple, banana,lemon)		q_1	blue
q_2: (toy, blue, orange)		q_1, q_2	blue
t_2: car		q_1, q_2	blue, car
t_3: orange		q_1, q_2	blue, car, orange
q_3: (orange, blue)	found in PC	q_1, q_2	blue, car, orange
t_4: mouse		q_1, q_2	blue, car, orange, mouse

FIGURE 4: Running Example of Algorithm SOF.

B. DISCUSSIONS

Three filling algorithms, i.e., QOF, TOF, and SOF are proposed to establish the dependency between RC and PC with the same purpose of reducing the duplicate hits in RC and PC. If considering a baseline filling algorithm that fills RC

and PC separately without considering their content, RC will cache the results of queries q_1, q_2, and q_3 according to the scores of the queries in Table 1 and PC will cache the posting lists of terms 'blue', 'car', 'orange', 'mouse', and 'toy' according to the scores of the terms in Table 2. Given two testing queries (toy, blue, orange) and (orange, blue), there exist duplicate hits in the cache filled by the baseline algorithm. However, no duplicate hit is found in the caches filled by our proposed algorithms.

Among the three proposed algorithms, TOF is the most effective algorithm, since it does not have any duplicate hit. Algorithms QOF and SOF may yield caches with duplicate hits.

C. TERM SET GENERATION

In the C3C architecture, the intersection cache (IC) stores the intersections of the posting lists for some term sets. Each term set contains several terms that have appeared in some queries. Previously, frequent itemset mining has been applied for generating term sets for the intersection cache [8]. However, this method only considers the frequency of the term sets. The space needed for storing the intersection of the posting lists is ignored. This section proposes a new algorithm that considers both the frequency and the space needed for caching when generating term sets. Algorithm 4 describes the pseudo code of term set generation algorithm TSG. It consists of the following three steps.

Algorithm 4 TSG

$TS \leftarrow \emptyset$
for *each distinct query* q *in the query log* Q **do**
  $S \leftarrow \mathbf{GenerateTermSets}(q)$
  for *each set* s *in* S **do**
    $s.size \leftarrow \mathbf{Space}(s)$
  end
  $s_m \leftarrow$ choose the set with the smallest $s.size$ in S
  $s_m.fre \leftarrow 0$
  Add s_m to TS
end
for *each query* q *in the query log* Q **do**
  $S \leftarrow \mathbf{GenerateTermSets}(q)$
  for *each set* s *in* S **do**
    if TS *contains an entry for* s **then**
      $s.fre++$
    end
  end
end
return TS

Step 1: **Initialization:** The generated term sets are stored in a list TS, where each element s represents a term set. Let $s.size$ be the space needed for storing the intersection of the posting lists of the terms in s and $s.fre$ be the frequency of the term set. In the beginning, TS is empty.

Step 2: **Generating Term Sets:** For each distinct query in the query log, algorithm TSG firstly generates all possible term sets from the query by calling function $\mathbf{GenerateTermSets}(q)$. In this algorithm, we only consider the term sets that contain no less than 2 terms. For each generated term set, the space needed for storing the intersection of the posting lists of the terms in the term set is calculated by calling function $\mathbf{Space}(s)$. Then the term set with the smallest space needed is added to TS and its frequency is initialized as 0.

Step 3: **Updating Frequency:** Having the generated term sets in TS from the previous step, algorithm TSG scans the query log and calculates the frequency of each term set in TS. Specifically, for each query in the query log, function $\mathbf{GenerateTermSets}(q)$ is called to generate all possible term sets from the query. For each generated term set, if it is found in TS, the frequency of the term set is increased by 1. After all the queries in the query log have been processed, algorithm TSG returns TS.

D. DEPENDENCIES AMONG RC, PC AND IC

This section presents the filling algorithm of the content-based three-level web cache (C3C). The C3C consists of three caches, i.e., RC, PC, and IC. In previous sections, three algorithms for establishing the dependency between RC and PC are proposed. According to empirical study in Section V, algorithm TOF outperforms the other two algorithms. Motivated by algorithm TOF, the C3C filling algorithm proposed in the section fills the three caches in the order of PC, IC, and RC. The cache to be filled later will consider the contents of the caches filled before. In other words, the content of IC depends on the content of PC and the content of RC depends on both the contents of PC and IC. The C3C filling algorithm tries to avoid cached duplicate data, so that the cache space is better used. Algorithm 5 describes the pseudo code of the C3C filling algorithm. It consists of the following four steps.

Step 1: **Scoring:** the algorithm calculates the score for each term, each generated term set and each query, by dividing its frequency by the space needed for caching it, i.e., $score = freq/size$. Terms, term sets, and queries are sorted in the descending order of their scores, respectively.

Step 2: **Filling PC:** PC is filled up with the posting lists of terms with high scores.

Step 3: **Filling IC:** This algorithm fills up IC with the intersection of the posting lists of the generated term sets with high scores. For each term set under consideration, if all the terms in this term set have been cached in PC, this term set is discarded. Otherwise, it is added to IC.

Step 4: **Filling RC:** RC is filled up with the results of queries with high scores. For each retrieved query, the algorithm first checks whether the union of some term sets cached in IC equals to the query. If yes, the query is discarded, otherwise, the missed terms

are calculated. Then the algorithm tries to find the missed terms in PC. If all missed terms are found, the query is discarded. Otherwise, the results of the query are added to RC.

Algorithm 5 C3C Filling Algorithm

```
Posting list cache PC ← ∅
Result cache RC ← ∅
Intersection Cache IC ← ∅
For each term, compute score_T ← freq_T / size_T
For each query, compute score_Q ← freq_Q / size_Q
For each term set, compute score_S ← freq_S / size_S
while PC is not full do
    t ← getNextTerm()
    add term t into PC
end
while IC is not full do
    s ← getNextTermSet(TS)
    if PC stored the posting list of all terms in s then
        continue
    else
        add s into IC
    end
end
while RC is not full do
    q ← getNextQuery()
    S ← GenerateTermSets(q)
    FoundTerm ← ∅
    foreach s ∈ S do
        if s is found in IC then
            Add s to FoundTerm
        end
    end
    if FoundTerm == q then
        continue
    else
        MissedTerm ← q − FoundTerm
        if PC stored the posting list of MissedTerm then
            continue
        else
            add the results of query q to RC
        end
    end
end
```

TABLE 3: Example Terms, Term Sets, and Queries.

Term List	Term Set List	Query List
t_1	$s_1 = \{t_5, t_6\}$	$q_1 = \{t_2, t_4, t_7\}$
t_2	$s_2 = \{t_2, t_3\}$	$q_2 = \{t_4, t_5, t_6, t_7\}$
t_3	$s_3 = \{t_4, t_7\}$	$q_3 = \{t_2, t_3, t_5\}$
t_4		

Example 4. Table 3 shows example lists of sorted terms, term sets, and queries. Fig. 5 illustrates how the C3C filling algorithm works. Firstly, PC is filled up with the posting lists of 4 terms t_1, t_2, t_3, t_4. Then the algorithm starts to fill IC.

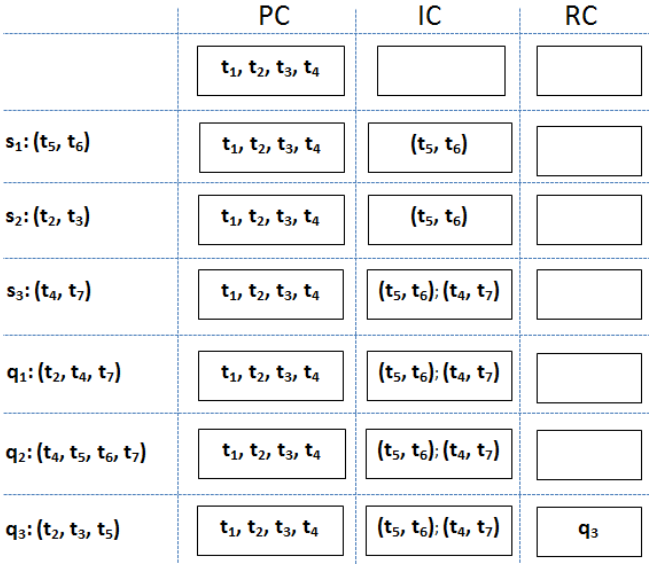


FIGURE 5: Running Example of the C3C Filling Algorithm.

For each term set in the sorted list, if all the terms are found in PC, the term set will not be cached in IC. In this example, term sets s_1 and s_3 are added to IC, while s_2 is discarded, since all the terms in s_2 are found in PC. Next, the algorithm starts to fill RC. For query q_1, term set $\{t_4, t_7\}$ is found in IC and term t_2 is found in PC, so q_1 is not cached in RC. For query q_2, there exist two term sets $\{t_4, t_7\}$ and $\{t_5, t_6\}$ in IC, such that the union set equals to q_2. Thus, q_2 is discarded. Query q_3 is added to RC, because there is no way to construct q_3 by only using the cached data in PC and IC.

V. EXPERIMENTS

A. EXPERIMENTAL SETUP

Data. A real AOL query log with 36 million queries issued in 2016, from March to May, was used in the experiments. Stopping words and punctuations in all queries were eliminated, and queries which were only issued onces were also removed. The cleaned query log was then sorted in the ascending order of the issued times of the queries. This log was divided into two parts: the first 70% for training and the second 30% for testing. A collection of around 8.8 million Wikipedia[2] documents was applied as the documents results to be queried in the experiments. These documents were indexed by using Apache Lucene[3]. Tables 4 and 5 summarize the detailed information of the data.

The top 50 results [13] were retrieved for each query in the cleaned query log to be stored in RC, and these results take about $30KB$. Each result covers all the keywords in the query and stores a title, an URL and a snippet. In general, the posting list of a given term that is stored in PC is a list of L entries. Each entry consists of a document identifier and a frequency of the term in the document, and it uses 8 bytes.

[2]https://dumps.wikimedia.org/enwiki/20161201
[3]https://lucene.apache.org

The size of the term's posting list in PC is $8 \times L$. We also set up the same entry of intersection list of an itemset as the posting list of a term.

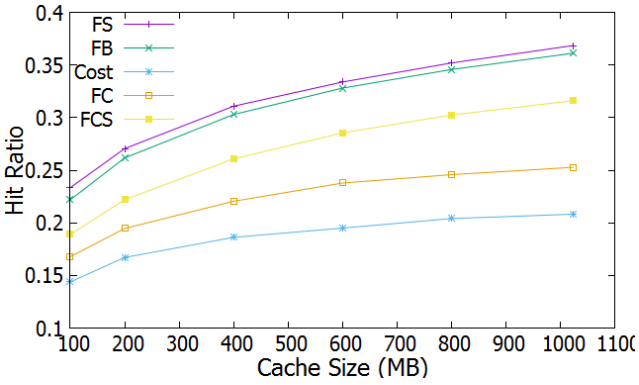
Platform. All algorithms were written in Java and executed on a Window 10 machine with 3.4 GHz dual-core CPU and 12 GB memory.

TABLE 4: Statistics of Cleaned AOL Query Log.

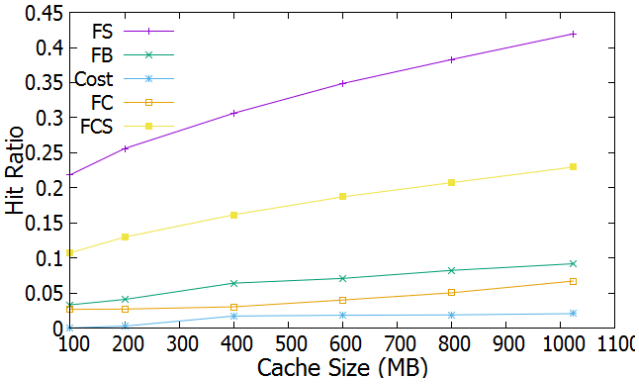
	Query Log	Training Set	Testing Set
Number of queries	24,201,747	16,941,221	7,260,526
Number of unique queries	2,989,815	2,289,475	1,195,689
Average length of queries	3.1		
The longest length of queries	46	46	33
Number of terms	314,831	274,969	190,216

TABLE 5: Statistics of Document Collection.

	Document Collection
Document Size	64.5 GB
Index Size	36.7 GB
Number of documents	8,859,888
Number of terms	57,903,421
Number of tokens	5,402,701,012



(a) Result Cache.



(b) Posting List Cache.

FIGURE 6: Hit Ratios of Five Caching Polices.

B. EVALUATING THE ALGORITHMS OF FILLING RC AND PC

Caching Polices. The five prevalent caching policies are reviewed in Section II, namely FB, FS, Cost, FC and FCS. In order to investigate one caching policy that improves the hit ratios in our experimental setting, an empirical study was conducted on the result cache (RC) and the posting list cache (PC). We applied the five polices to both the two caches. Fig. 6 plots the hit ratios as a function of cache size for the five caching policies. Consistently, policy FS beats the other polices on both RC and PC. Hence, FS is taken as the default caching policy in the following experiments.

The Hit Ratios. We report the hit ratios of the proposed three caching filling algorithms, i.e., QOF, TOF, and SOF and compare them with a *baseline* algorithm that fills *RC* and *PC* separately. The filling algorithm of *RC* and *PC* proposed in a *five-level* cache architecture [17] is also taken as a comparison. Fig. 7 illustrates the hit ratios achieved by the five methods when the total cache sizes of $200MB$ and $500MB$ were used. The x-axis presents the ratio of the cache space of RC to the cache space of PC. We can observe that the three proposed algorithms perform better than the comparing methods and Algorithm TOF achieves the best performance. When the space size of RC varies from 10% to 40%, the hit ratio increases. However, it decreases as the RC's space size changes from 50% to 90%. Fig. 8 presents the duplicate hit ratios resulted from the five methods. Because TOF produces 0 duplicate hit ratio from caching, this algorithm is the best one.

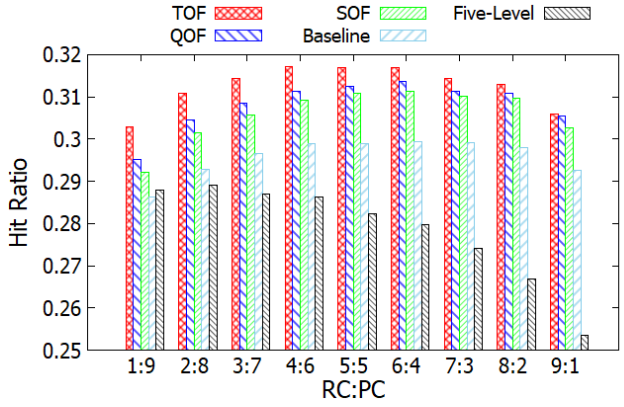
C. EVALUATION OF C3C

In the previous experiments, it has been shown that algorithm TOF achieves the best performance in the dependency establishment between RC and PC. Hence, the proposed C3C filling algorithm adopts the idea of algorithm TOF. In this section, we evaluate the performance of C3C where dependencies are established among RC, PC, and IC.

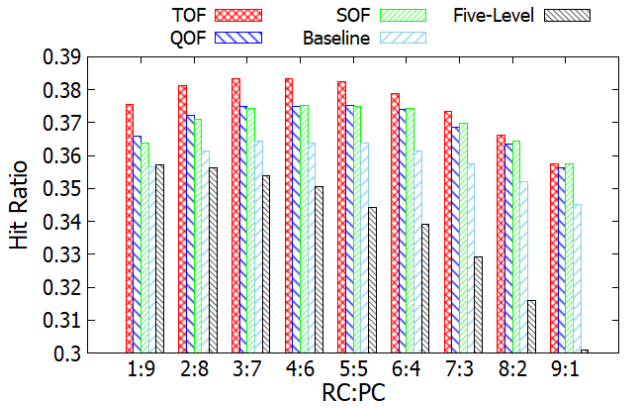
Term Set Generation. We generate two-term and three-term sets using algorithm TSG and evaluate the following four variants of IC in C3C architecture.

- **IC3**: only the intersections of the posting lists of three-term sets are cached.
- **IC2**: only the intersections of the posting lists of two-term sets are cached.
- **IC23**: both the intersections of the posting lists of two-term and three-term sets are cached. The term sets are chosen according to strategy FS described in Section II.
- **IC2-IC3**: IC is split into two parts. One part caches the intersections of the posting lists of two-term sets. The other part caches the intersections of the posting lists of three-term sets.

Hit Ratios Comparisons. Fig. 9 shows the hit ratio of IC2-

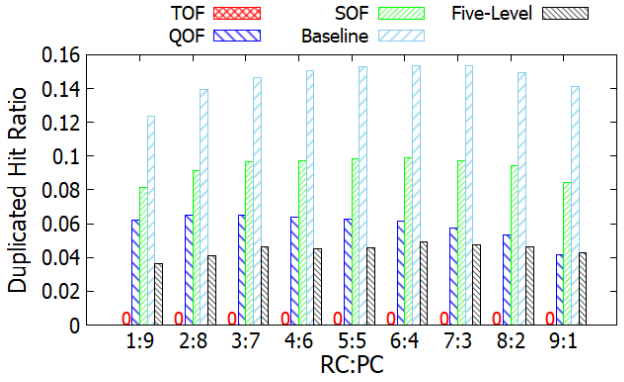


(a) Cache Size 200 MB.

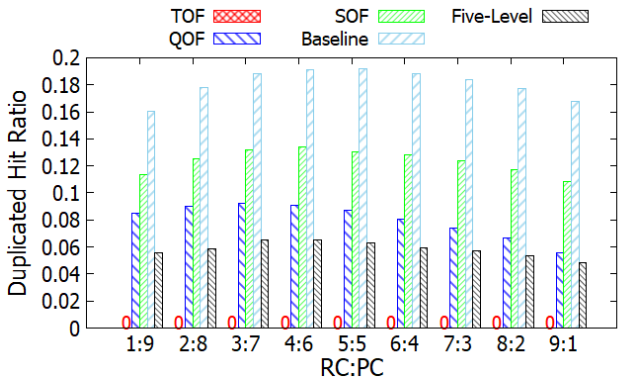


(b) Cache Size 500 MB.

FIGURE 7: Hit Ratios of Five Methods.



(a) Cache Size 200 MB.



(b) Cache Size 500 MB.

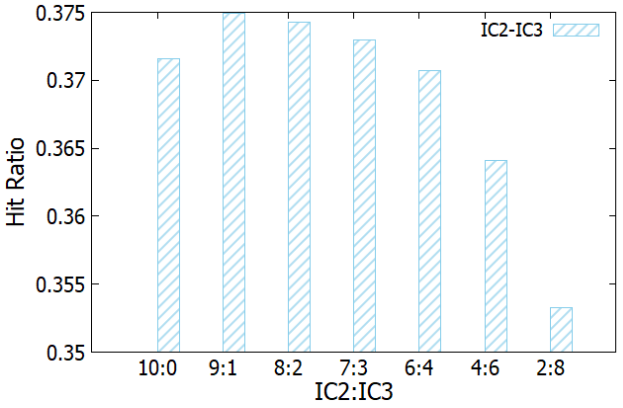
FIGURE 8: Duplicate Hit Ratios of Five Methods.

IC3 when varying the ratio of the sizes of IC2 and IC3 in an intersection cache. We observe that when the size of IC2 is set to 90% and the size of IC3 is set to 10%, the hit ratio of the intersection cache is the highest. In the following experiments, we fix the ratio of the sizes of IC2 and IC3 at 90% to 10% for IC2-IC3 intersection cache. Fig. 10 shows the hit ratios of C3C where IC takes the form of IC2, IC3, IC23, and IC2-IC3. The x-axis is the ratio of the sizes of RC, PC, and IC in C3C. The total size of C3C is set to $200MB$ and $500MB$. We observe that IC2-IC3 achieves the best performance compared with other variants of the intersection cache. The best way of allocating space for RC, PC, and IC is 30%, 20%, and 50%. Fig. 11 and Fig. 12 show the hit ratios and the duplicate hit ratios of C3C, Five-Level cache [17], and TLMCA [8]. C3C is the best in terms of both the hit ratio and the duplicate hit ratio. The reason is that the other two caches do not consider the content stored in each part, so that the cache space is under utilized, resulting in duplicate hits.

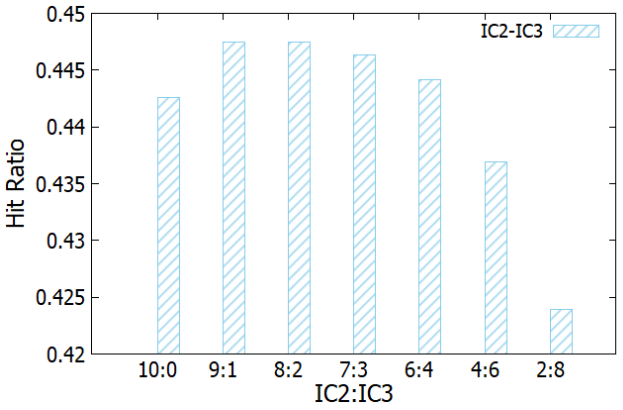
Query Processing Time. We evaluate the performance of the three methods (C3C, Five-Level and TLMCA) in terms of the total query processing time. Fig. 13 shows the total query processing time of the three methods versus the two cache sizes, $200MB$ and $500MB$. As expected, avoiding duplicated contents of the three parts (RC, PC and IC) also improves the performance of C3C about the total query processing time. We observe that C3C is more efficient than the two existing methods. In other words, the total query processing time achieved by C3C is better than that of Five-Level, and TLMCA is the worst.

VI. CONCLUSION

In this work, we have proposed the new static content-based three-level web cache (C3C) that consists of three parts, i.e., a result cache, a posting list cache, and an intersection cache. In order to better use the cache space, we propose the C3C filling algorithm that establishes dependencies among the three parts. In other words, what to be cached in one part depends on the content of another parts, so that the three parts will not cache the same content. By using the proposed algorithm, the duplicate hits occurring in the existing methods are reduced and the hit ratio increases. We also propose a term set generation algorithm that provides the content to be cached in the intersection cache. The C3C filling algorithm takes the term sets as the source of the intersection cache. Extensive experiments were conducted on a real data set. The results prove the effectiveness of the proposed cache architecture and the corresponding filling algorithms.

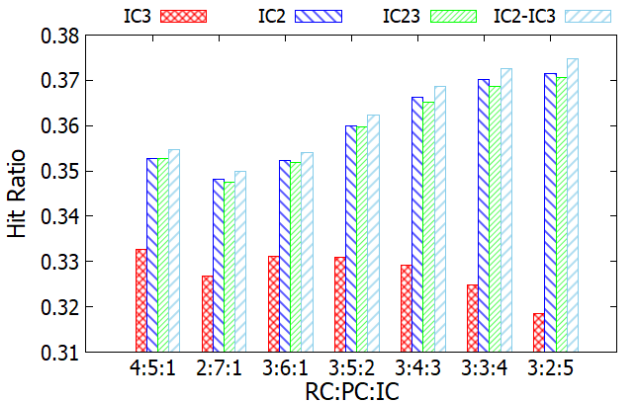


(a) Cache size 200 MB.

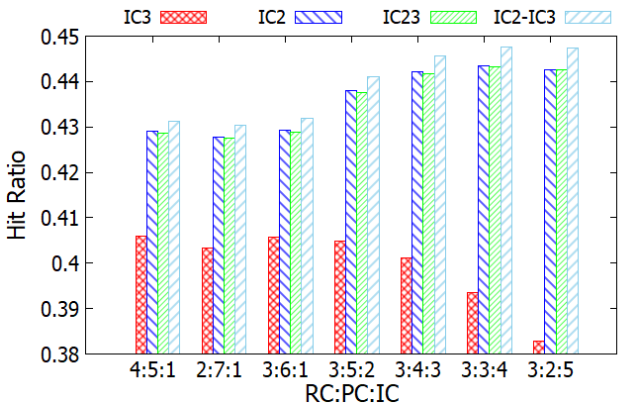


(b) Cache size 500 MB.

FIGURE 9: Hit ratios of C3C used IC2-IC3, cache space allocated RC, PC and IC is 30%, 20% and 50% respectively.



(a) Cache size 200 MB.



(b) Cache size 500 MB.

FIGURE 10: Hit ratios of C3C used IC3, IC2, IC23 and IC2-IC3.

REFERENCES

[1] E. P. Markatos, "On Caching Search Engine Query Results," Computer Communications, vol. 24, no. 2, pp. 137–143, 2001.

[2] 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} Trans. Inf. Syst., vol. 24, no. 1, pp. 51–78, 2006.

[3] I. S. Altingövde, R. Ozcan, and Ö. Ulusoy, "A Cost-Aware Strategy for Query Result Caching in Web Search Engines," in ECIR, 2009, pp. 628–636.

[4] Q. Gan and T. Suel, "Improved techniques for result caching in web search engines," in WWW, 2009, pp. 431–440.

[5] R. Ozcan, I. S. Altingovde, and Ö. Ulusoy, "Static query result caching revisited," in Proceeding of the 17th international conference on World Wide Web - WWW '08. New York, New York, USA: ACM Press, 2008, p. 1169. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1367497.1367710

[6] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri, "The impact of caching on search engines," in Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR '07. New York, New York, USA: ACM Press, 2007, p. 183. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1277741.1277775

[7] J. Zhang, X. Long, and T. Suel, "Performance of Compressed Inverted List Caching in Search Engines," Refereed Track: Search - Corpus Characterization & Search Performance, pp. 387–396, 2008.

[8] W. Zhou, R. Li, X. Dong, Z. Xu, and W. Xiao, "An Intersection Cache Based on Frequent Itemset Mining in Large Scale Search Engines," in IEEE Workshop on Hot Topics in Web Systems and Technologies, 2015, pp. 19–24.

[9] G. Tolosa, L. Becchetti, E. Feuerstein, and A. Marchetti-Spaccamela, "Performance Improvements for Search Systems Using an Integrated Cache of Lists+Intersections," in SPIRE, 2014, pp. 227–235.

[10] D. Ceccarelli, C. Lucchese, S. Orlando, R. Perego, and F. Silvestri, "Caching query-biased snippets for efficient retrieval," Proceedings of the 14th International Conference on Extending Database Technology, pp. 93–104, 2011.

[11] R. Ozcan, I. S. Altingövde, B. B. Cambazoglu, and Ö. Ulusoy, "Second Chance: {A} Hybrid Approach for Dynamic Result Caching and Prefetching in Search Engines," TWEB, vol. 8, no. 1, pp. 3:1—-3:22, 2013.

[12] S. Podlipnig and L. Böszörményi, "A survey of Web cache replacement strategies," {ACM} Comput. Surv., vol. 35, no. 4, pp. 374–398, 2003.

[13] P. C. Saraiva, E. Silva de Moura, N. Ziviani, W. Meira, R. Fonseca, and B. Riberio-Neto, "Rank-preserving two-level caching for scalable search engines," in Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR '01. New York, New York, USA: ACM Press, 2001, pp. 51–58. [Online]. Available: http://dl.acm.org/citation.cfm?id=383952.383959 http://portal.acm.org/citation.cfm?doid=383952.383959

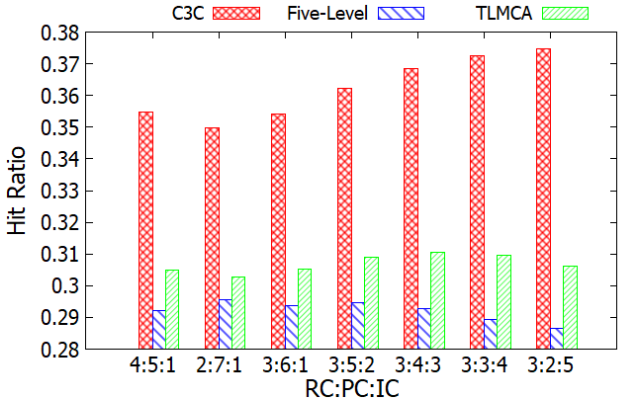
[14] R. A. Baeza-Yates and S. Jonassen, "Modeling Static Caching in Web Search Engines," in ECIR, 2012, pp. 436–446.

[15] X. Long and T. Suel, "Three-Level Caching for Efficient Query Processing in Large Web Search Engines," World Wide Web, vol. 9, no. 4, pp. 369–395, dec 2006.

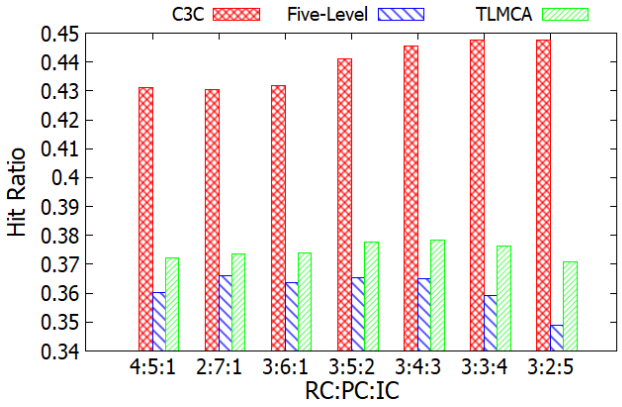
[16] M. Marin, V. G. Costa, and C. Gómez-Pantoja, "New caching techniques for web search engines," in HPDC, 2010, pp. 215–226.

[17] R. Ozcan, I. S. Altingövde, B. B. Cambazoglu, F. P. Junqueira, and Ö. Ulusoy, "A five-level static cache architecture for web search engines," Inf. Process. Manage., vol. 48, no. 5, pp. 828–840, 2012.

[18] X. Dong, R. Li, H. He, X. Gu, M. Sarem, M. Qiu, and K. Li, "EDS: An Efficient Data Selection policy for search engine storage

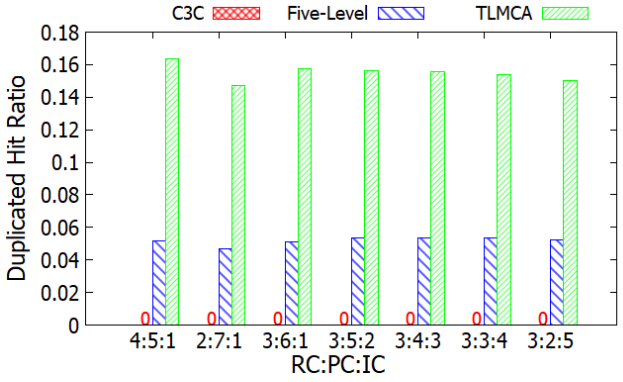


(a) Cache size 200 MB.

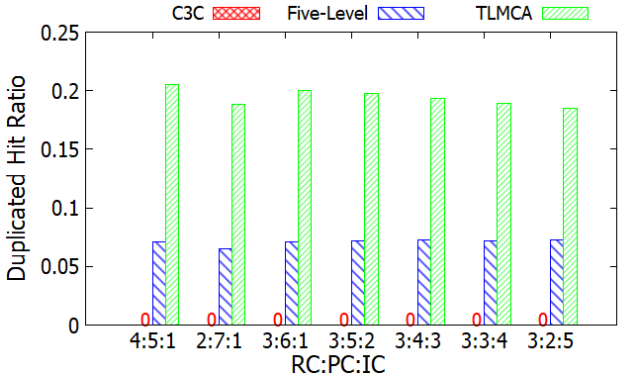


(b) Cache size 500 MB.

FIGURE 11: Hit Ratio.



(a) Cache size 200 MB.



(b) Cache size 500 MB.

FIGURE 12: Duplicate Hit Ratio.

architectures," Future Generation Computer Systems, 2015. [Online]. Available: http://dx.doi.org/10.1016/j.future.2016.02.014

[19] A. Detti, L. Bracciale, P. Loreti, and N. B. Melazzi, "Modeling LRU cache with invalidation," Computer Networks, vol. 134, pp. 55–65, 2018. [Online]. Available: https://doi.org/10.1016/j.comnet.2018.01.029

[20] R. Baeza-Yates, A. Gionis, F. P. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri, "Design trade-offs for search engine caching," ACM Transactions on the Web, vol. 2, no. 4, pp. 1–28, 2008.

[21] T. Kucukyilmaz, B. B. Cambazoglu, C. Aykanat, and R. Baeza-Yates, "A machine learning approach for result caching in web search engines," Information Processing and Management, vol. 53, no. 4, pp. 834–850, 2017.

[22] J. Wang, E. Lo, M. L. Yiu, J. Tong, G. Wang, and X. Liu, "The impact of solid state drive on search engine cache management," Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval - SIGIR '13, p. 693, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2484028.2484046

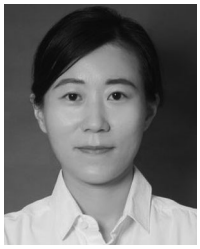
[23] T. Ma, Y. Hao, W. Shen, Y. Tian, and M. Al-Rodhaan, "An Improved Web Cache Replacement Algorithm Based on Weighting and Cost," IEEE Access, vol. 6, pp. 27 010–27 017, 2018.

[24] G. Hasslinger, K. Ntougias, F. Hasslinger, and O. Hohlfeld, "Performance evaluation for new web caching strategies combining LRU with score based object selection," Computer Networks, vol. 125, pp. 172–186, 2017. [Online]. Available: http://dx.doi.org/10.1016/j.comnet.2017.04.044

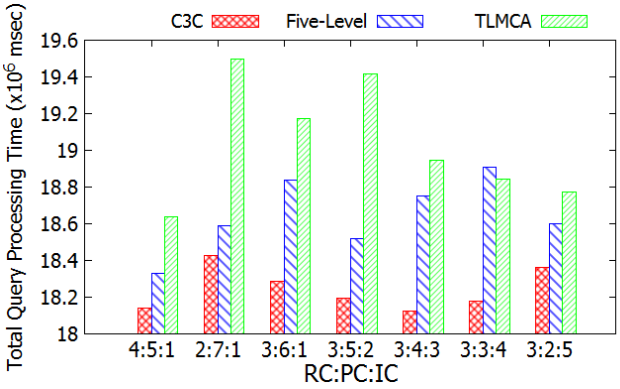
[25] N. Bechmann, H. Chen, and A. Cidon, "LHD: Improving Cache Hit Rate by Maximizing Hit Density," USENIX Symposium on Networked Systems Design and Implementation, no. 18, pp. 389–403, 2018.

[26] R. Ozcan, I. S. Altingövde, and Ö. Ulusoy, "Cost-Aware Strategies for Query Result Caching in Web Search Engines," TWEB, vol. 5, no. 2, pp. 9:1—-9:25, 2011.

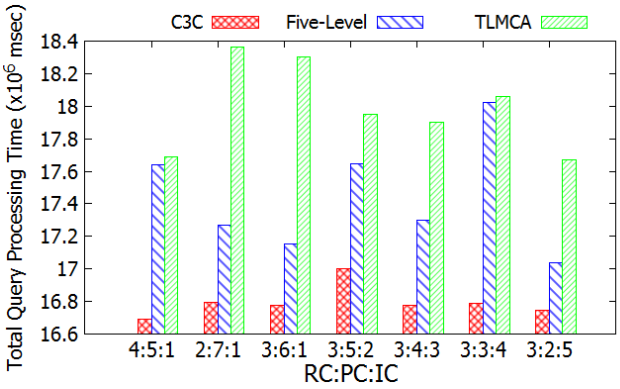
THANH TRINH is from Vietnam. He received the MSc degree in Information Systems Design from University of Central Lancashire, UK. He is currently pursuing the Ph.D. degree with Shenzhen university, China. He has authored several conference papers on his research topic. His research includes efficient query, database, social network, classification, forecasting disasters.



DINGMING WU received the PhD degree in computer science from Aalborg University, Denmark, in 2011. She is an assistant professor in the College of Computer Science & Software Engineering, Shenzhen University, China. Her general research area is data management and mining, including data modeling, database design, and query languages, efficient query and update processing, indexing, and mining algorithms.



(a) Cache size 200 MB.



(b) Cache size 500 MB.

FIGURE 13: Total Query Processing Time.

JOSHUA ZHEXUE HUANG received the PhD degree from the Royal Institute of Technology, Sweden. He is now a professor in the College of Computer Science and Software, Shenzhen University, and professor and chief scientist in the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, and honorary professor in the Department of Mathematics, The University of Hong Kong. His research interests include data mining, machine learning, and clustering algorithms.