Diversified Caching for Replicated Web Search Engines

Chuanfei Xu, Bo Tang, Man Lung Yiu

Department of Computing
The Hong Kong Polytechnic University
{csxchuanfei, csbtang, csmlyiu}@comp.polyu.edu.hk

***Abstract*— Commercial web search engines adopt parallel and replicated architecture in order to support high query throughput. In this paper, we investigate the effect of caching on the throughput in such a setting. A simple scheme, called uniform caching, would replicate the cache content to all servers. Unfortunately, it does not exploit the variations among queries, thus wasting memory space on caching the same cache content redundantly on multiple servers. To tackle this limitation, we propose a diversified caching problem, which aims to diversify the types of queries served by different servers, and maximize the sharing of terms among queries assigned to the same server. We show that it is NP-hard to find the optimal diversified caching scheme, and identify intuitive properties to seek good solutions. Then we present a framework with a suite of techniques and heuristics for diversified caching. Finally, we evaluate the proposed solution with competitors by using a real dataset and a real query log.**

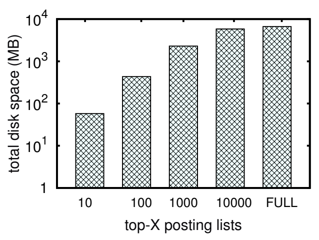
I. Introduction

Commercial web search engines [7], [21], [35] adopt the parallel architecture in order to support high query throughput. In this architecture, the broker receives queries from users, and then assigns those queries to servers for processing (cf. Figure 2). In addition, such parallel search engines apply replication [22], [23] to further enhance the query throughput. For example, the Google search engine adopts a parallel and replicated architecture [7].

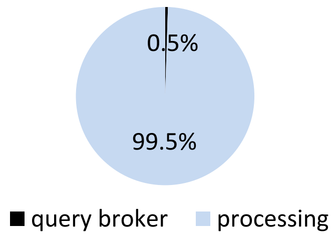
In the replicated architecture [10], both the document collection and the inverted index are replicated on multiple servers, where each *server* can be a single machine or a sub-cluster. We note that commercial web search engines like Google [7] and Baidu [21] replicate the entire inverted index on each server. This architecture scales well with the query throughput because it enables each server to process queries locally (by using its replicated data and index), without heavy communication across multiple servers. An experimental evaluation [10] has shown that this architecture is better than distributed architectures (that do not use replication). In addition, it offers high availability to cope with server failures.

Although full replication (i.e., the entire inverted index) leads to higher storage space, it is still acceptable in practice. Figure 1(a) shows the storage space for replicating only top-X (most frequent terms') posting lists, for a 75 GB real ClueWeb dataset[1]. The storage space of full replication is only slightly higher than the storage space of top-10000 posting lists. Furthermore, the huge capacity of modern hard disks renders it feasible to replicate inverted indexes. The maintenance overhead is also not an issue as search engines are query-intensive and the updates are usually executed in batches periodically.

[1]http://lemurproject.org/clueweb09



(a) disk space



(b) query time breakdown

Fig. 1: Replication on ClueWeb posting lists

The requirement for high query throughput is becoming more significant in commercial search engines. In this paper, we adopt the parallel search architecture with full replication [10] because (i) the query throughput scales well with the number of servers, and (ii) it avoids heavy communication across servers. Our goal is to investigate the effect of caching on the query throughput under such setting. Let n be the number of servers, S_i be the i-th server, and C_i be the cache content of S_i. Given a query workload $\mathcal{Q}$ (i.e., a sequence of queries), we measure the query throughput of this system as $\frac{|\mathcal{Q}|}{T_{total}(\mathcal{Q})}$, where the total processing time $T_{total}(\mathcal{Q})$ is defined as:

$$T_{total}(\mathcal{Q}) = \max\{\, T_{C_1}(Q_1), T_{C_2}(Q_2), \cdots, T_{C_n}(Q_n)\,\},$$

with $Q_i \subset \mathcal{Q}$ as a subset of queries assigned to server S_i, and $T_{C_i}(Q_i)$ as the processing time of Q_i by using C_i (on S_i). As shown in Figure 1(b), the majority of time is spent on processing at servers, rather than at the broker. Thus, we ignore the broker time in $T_{total}(\mathcal{Q})$. Our objective is to minimize $T_{total}(\mathcal{Q})$ in order to maximize the query throughput. This leads to two subproblems: (i) deciding the cache content C_i of each server, (ii) deciding the subset of workload Q_i for each server.

To our best knowledge, existing caching techniques [4], [18], [25] have not considered the above architecture and exploited its optimization opportunities. A simple scheme, which we call *uniform caching*, would replicate the cache content to all servers. Suppose that the cache content are posting lists for frequently-used terms (e.g., PL_{ipad} in Table I). Unfortunately, uniform caching does not exploit the variations of terms among queries, so it may waste memory space on caching redundant posting lists on multiple servers. We illustrate this in the following example, and model the processing time by its dominant factor —— the number of disk seeks [2].

To tackle the above drawback, we propose a *diversified caching* problem, which aims to diversify the types of queries served by different servers, and maximize the sharing of terms among queries assigned to the same server.

We illustrate our problem as follows. Table I(a) shows a document collection and the corresponding *inverted index* [38], in which each posting list PL_j records the IDs of documents containing term j (see Table I(b)). Suppose that the replicated system contains two servers (S_1, S_2); each server stores all PL_j in disk and keeps a cache for PL_j (in RAM) with a fixed capacity (e.g., 3 document IDs). The query workload is shown in Table I(c).

TABLE I: Example of documents, posting lists, and queries

d_1	ipad, apple
d_2	galaxy
d_3	iphone, galaxy
d_4	ipad, apple, gear
d_5	ipad

(a) document collection

PL_{iphone}	d_3
PL_{ipad}	d_1, d_4, d_5
PL_{apple}	d_1, d_4
PL_{galaxy}	d_2, d_3
PL_{gear}	d_4

(b) posting lists

q_1	ipad, apple
q_2	gear, iphone
q_3	galaxy
q_4	ipad, iphone

(c) query workload

TABLE II: Caching schemes for a replicated search system

	(a) uniform scheme		(b) diversified scheme	
Server	S_1	S_2	S_1	S_2
Cache C_i (in RAM)	PL_{ipad} 3	PL_{ipad} 3	PL_{ipad} 3	PL_{galaxy} 2 PL_{gear} 1
Disk	all PL_*	all PL_*	all PL_*	all PL_*
Queries Q_i & disk seeks	q_1: 1 q_3: 1	q_2: 2 q_4: 1	q_1: 1 q_4: 1	q_2: 1 q_3: 0
Throughput	$1.33 = 4/\max\{2,3\}$		$2 = 4/\max\{2,1\}$	

This system has 2 servers; cache capacity is 3 (document IDs)

Example for uniform caching: Table II(a) demonstrates the uniform caching scheme. Since all caches have the same content, all servers incur the same processing cost for the same query. In order to reduce $T_{total}(\mathcal{Q})$, we balance $T_{C_i}(Q_i)$ among servers by assigning queries to servers in the round-robin fashion. For instance, server S_1 receives a query q_1 with two terms: 'ipad' and 'apple'. Since PL_{ipad} is in the cache, it incurs only 1 disk seek to fetch PL_{apple} from the disk. The total costs on servers S_1 and S_2 are 1+1 and 2+1 disk seeks, respectively. So the query throughput is: $4/\max\{2,3\} = 1.33$.

Example for diversified caching: Table II(b) shows an example for our scheme. It may cache different posting lists on different servers. As such, different servers may incur different processing costs for the same query. For example, it is cheaper to execute q_3 on server S_2. Regarding the assignment of queries, an intuitive strategy is to assign each query to the server with the cheapest cost. Observe that the query throughput is: $4/\max\{2,1\} = 2$, which is better than the uniform caching scheme.

Our research problem is to find the optimal *diversified caching scheme* that incurs low $T_{total}(\mathcal{Q})$ (thus providing high query throughput). We should consider not only the cache content C_i of each server, but also possible subsets of the workload Q_i assigned to different servers. Current partitioning techniques [2], [11], [12], [19], [37], either document-wise or term-wise, are designed to partition documents or the inverted index in a distributed architecture. They do not examine the coupling between C_i and Q_i as in our scenario. Also, existing load balancing techniques [23], [30] have not exploited the cache content C_i to further optimize the assignment of queries to servers.

Our problem is challenging as the combinations between C_i and Q_i on all n servers lead to a huge search space: $O\left(\frac{n^{|\mathcal{Q}|}}{n!} \cdot \frac{n^{|W|}}{n!}\right)$, where n, $|\mathcal{Q}|$, $|W|$ are the numbers of servers, queries, and distinct terms, respectively. Nevertheless, we discover an interesting characteristic in a real query workload $\mathcal{Q}$ (i.e., in AOL query log) that brings us opportunities for solving our problem more effectively. Specifically, we observe that two frequent terms w_i, w_j in Q may not co-occur frequently. We verify this on a real query log $\mathcal{Q}$ used in the experimental study. We picked top-8 most frequent terms (in $\mathcal{Q}$) in descending order, and reported the conditional probability $Pr(w_i, w_j \in q | w_i \in q)$ for each pair in Table III. For example, w_1 and w_5 do not co-occur frequently. Thus, it is desirable to cache these two frequent terms on different servers. Unfortunately, existing caching methods fail to consider relations among terms. In contrast, our proposed *Diversified Caching* (DC) schemes can handle such case in a better way (A more detailed review of this method appears in Section IV).

TABLE III: $Pr(w_i, w_j \in q | w_i \in q)$ on AOL query log

w_i/w_j	w_2	w_3	w_4	w_5	w_6	w_7	w_8
w_1	0.00	0.006	0.27	0.00	0.00	0.00	0.26
w_2		0.00	0.00	0.00	0.00	0.48	0.00
w_3			0.00	0.00	0.00	0.00	0.00
w_4				0.00	0.41	0.00	0.43
w_5					0.00	0.00	0.00
w_6						0.00	0.54
w_7							0.00

We summarize our technical contributions in this paper as follows.

- We formulate our diversified caching problem on the replicated search architecture, and prove this problem is NP-hard (Section III).
- We propose a framework with a suite of techniques and heuristics for diversified caching (Section IV).
- We evaluate the effectiveness of our proposed solution through extensive experiments on real datasets (Section V).

The rest of the paper is organized as follows. Section II summarizes the related work. Section III formally defines the problem and the model in this paper. Section IV presents our solution. Section V illustrates the experimental results. Section VI concludes the paper.

II. Related Work

Inverted index organization and replication: There have been extensive research [9], [11], [19], [29], [31], [37] on organizing inverted index in distributed or parallel search engines. These works on sharding inverted lists can be summarized in two categories: document-wise based [11], [37] and term-wise based methods [12], [13]. The work by Cambazoglu et al. [12] presents a comparison between term-wise and document-wise partitioned indexes using multiple servers. The experimental results show that, term-wise partitioned indexes give higher throughput, but also longer query latency. For these inverted index organization methods, each server only stores a part of inverted index so that the relevant caching techniques in these works are not suitable for our problem.

Several works have devoted to replication techniques [7], [8], [10], [15], [23] in distributed systems. In [10], [23], replicated inverted lists are used to speedup keyword queries. Furthermore, Moffat et al. [23] selectively replicate the inverted lists of a number of high workload terms, potentially halving the peak workload associated with each copy of the list. In [10], the authors compare replicated systems with sharding systems, and their experimental results show replicated systems have higher throughput. Nevertheless, these existing replication papers do not consider the cache content C_i to further optimize the assignment of queries to servers on the replicated search systems. In contrast, our work consider the coupling between cache content C_i and the subset of workload Q_i.

Query assignment: In distributed or parallel systems, we need to allocate queries to different machines. This process is called query assignment. For existing query assignment methods, Puppin et al. [28] first partition queries based on similar query results. Each group stores the text of each query belonging to a group, as a single text file. When a new query is submitted, they use the TF-IDF metric to find the best matches. This way, each query is assigned to a group based on the scores computed by TF-IDF metric. In [24], the authors propose a vertical partitioning of data (i.e., inverted lists) based on statistics. If two terms co-occur frequently, this method would put them into a group. Additionally, in replicated databases, Consens et al. [15] present an effective method to partition SQL queries. This work uses existing tools (e.g., DB2's Design Advisor) to compute the corresponding configurations and allocate queries to servers based on query costs. This method can be adapted to our problem setting; however, it does not achieve a high query throughput in our setting. The reasons are: (i) their query assignment methods tend to assign each query to the server with the lowest processing cost, but ignore the loads of severs; and (ii) they lack an online load balancing to balance I/O costs on different servers. In contrast, we consider the loads of servers during query allocation, and we propose an effective online load balancing method for servers.

III. Problem Statement

We first introduce the architecture of parallel and replicated search engine [3]. Next, we formulate our diversified caching problem. Then, we show the hardness of our problem. Table IV summarizes the symbols to be used in this paper.

TABLE IV: Commonly used symbols

Symbol	Description
n	number of servers
$\mathcal{B}$	cache capacity
w_s	a term
PL_s	posting list of w_s
S_i	i-th server
C_i	set of terms (posting lists) cached at S_i
$\mathcal{Q}$	query workload
$\mathcal{Q}^{train}$, $\mathcal{Q}^{test}$	training / testing query workload
Q_i	subset of workload (sub-workload) assigned to S_i
q_j	a query
$\mathbf{w}(q_j)$	set of terms of q_j

A. Replicated Search Engine Architecture

We adopt the architecture of the replicated search engine, as shown in Figure 2. The *broker* receives queries and then assigns them to *servers* (in the bottom layer) for processing. Each server S_i employs an in-memory cache C_i to reduce the disk access cost. In this paper, we assume C_i is a posting list cache (i.e., keeping only posting lists) as this cache type has been studied extensively [4]–[6]. We leave the extension of our techniques for other cache types (e.g., intersection cache [20], document cache [34], snippet cache [14]) as future work.

We illustrate the steps in query processing in Figure 2. Upon receiving a query q_j, the broker forwards it to a server S_i based on an *assignment policy* [STEP ①]. The server S_i checks whether the terms of q_j (i.e., $w_{j1}, w_{j2}, ..., w_{jm}$) reside in its cache [STEP ②], and then fetches those missing posting list(s) from the local disk [STEP ③]. Finally, it can compute the top-k relevant documents $(d_{j1}, d_{j2}, ..., d_{jk})$ and return them to the broker [STEP ④].

In the web search caching literature [4], [6], [10], the training query workload $\mathcal{Q}^{train}$ denotes the historical query log, and the testing query workload $\mathcal{Q}^{test}$ denotes the queries received online. We observe that $\mathcal{Q}^{train}$ and $\mathcal{Q}^{test}$ have similar distribution. This is desirable for the static caching policy [5], [25], i.e., the cache content is determined offline (by using $\mathcal{Q}^{train}$) and it remains unchanged during online operations. Its advantage is that it avoids the cache maintenance overhead.

Thus, we adopt the static caching policy in this paper. Specifically, our problems include:

- The (offline) cache admission problem: decide the cache content of servers by using $\mathcal{Q}^{train}$;
- The (online) query assignment problem: assign a query q (in $\mathcal{Q}^{test}$) to a server S_i.

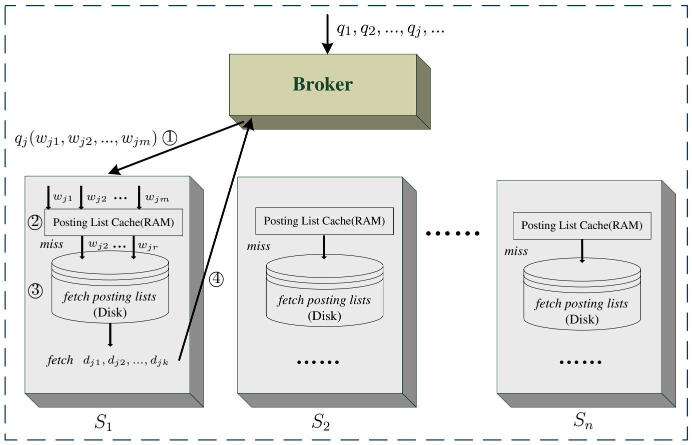

$q_1, q_2, ..., q_j, ...$
Broker
$q_j(w_{j1}, w_{j2}, ..., w_{jm})$ ①
w_{j1} w_{j2} ... w_{jm}
② Posting List Cache(RAM)
miss
w_{j2} ... w_{jr}
③ fetch posting lists (Disk)
④
fetch $d_{j1}, d_{j2}, ..., d_{jk}$
Posting List Cache(RAM)
miss
fetch posting lists (Disk)
......
S_1
S_2
S_n

Fig. 2: Architecture of the replicated search engine

B. Problem Definition

We proceed to review basic concepts for our problem on the above architecture.

*Definition 1 (**Cache**):* Let C_i be the (posting list) cache of the server S_i. We model C_i as a set of terms. The cache size of C_i is defined as $\sum_{w_s \in C_i} |PL_{w_s}|$, where $|PL_{w_s}|$ is the posting list size for the term w_s.

*Definition 2 (**Query and Workload**):* Let q_j be a query, $\mathbf{w}(q_j)$ denote the set of terms in q_j. A query workload Q is defined as a sequence of queries (in the order of submission time).

Recall that we use the training workload $\mathcal{Q}^{train}$ in the offline problem. We proceed to define the total processing time for $\mathcal{Q}^{train}$. The query broker partitions $\mathcal{Q}^{train}$ into n disjoint subsets $Q_1, Q_2, \cdots, Q_n$, where Q_i is the sub-workload assigned to server S_i. As in [2], the query processing time is dominated by the number of disk seeks. Thus, we define the processing time for Q_i on server S_i as:

$$T_{C_i}(Q_i) = \sum_{q_j \in Q_i} |\mathbf{w}(q_j) - C_i| \tag{1}$$

where $\mathbf{w}(q_j) - C_i$ represents the set of query terms in q_j but not in the cache C_i, leading to disk seeks.

As stated in the introduction, the query throughput can be expressed as the reciprocal of the total processing time for a workload. In our replicated architecture, the total processing time for $\mathcal{Q}^{train}$ is the maximum processing time among n servers:

$$T_{total}(\{\mathcal{Q}^{train}\}) = \max\{\ T_{C_1}(Q_1), T_{C_2}(Q_2), \cdots, T_{C_n}(Q_n)\ \}. \tag{2}$$

The rationale of diversified caching (DC) is to allow each server's cache to have different cache contents, tailored to a particular subset of the query workload. Our diversified caching problem consists of the following two subproblems.

*Problem 1 (**DC: Offline Cache Admission**):* Given a training set $\mathcal{Q}^{train}$, the cache capacity $\mathcal{B}$, and the number of servers n, the cache admission problem is to find a set of pairs

$$\{\langle Q_1, C_1\rangle, \langle Q_2, C_2\rangle, \cdots, \langle Q_n, C_n\rangle\}$$

such that the value $T_{total}(\{\mathcal{Q}^{train}\})$ is minimized, and each cache size is bounded by $\mathcal{B}$, i.e., $\sum_{w_s \in C_i} |PL_{w_s}| \leq \mathcal{B}$.

*Problem 2 (**DC: Online Query Assignment**):* Given a testing set $\mathcal{Q}^{test}$, the number of servers n, and their caches $C_1, C_2, \cdots, C_n$, the query assignment problem is to assign each query $q \in \mathcal{Q}^{test}$ to a server such that the value $T_{total}(\{\mathcal{Q}^{test}\})$ is minimized, where Q_i denotes i-th subset of $\mathcal{Q}^{test}$ assigned to server S_i.

For the online query assignment problem, the broker is not allowed to examine queries in future. Thus, it may only process queries in $\mathcal{Q}^{test}$ one-by-one.

C. Hardness of the Cache Admission Problem

First, we show that our offline cache admission problem is NP-hard.

Lemma 1: Our offline cache admission problem is NP-hard.

Proof: We prove this lemma by reducing the n-PARTITION problem [16] to our problem.

n-PARTITION problem [16]: A problem instance is $\langle \mathcal{U}, n\rangle$, where $\mathcal{U}$ is a set of positive integers, and n is an integer. This problem asks whether there exists a partitioning $\{U_1, U_2, ..., U_n\}$ of $\mathcal{U}$, such that $\forall 1 \leq i \leq n, \sum_{e \in U_i} e = \mathcal{X}$, where $\mathcal{X} = (\sum_{e \in \mathcal{U}} e)/n$. This problem is shown to be NP-hard [16].

Our simplified problem (n-CACHE): For the sake of NP reduction, we consider the decision version of our problem in Definition 1, and simplify it by fixing: (i) $\mathcal{B} = 0$ (i.e., each cache C_i is empty), and (ii) $|PL_s| = 1$. A problem instance is $\langle \mathcal{Q}^{train}, n\rangle$, where $\mathcal{Q}^{train}$ is query workload, and n is the number of servers. This problem (the decision version) asks for a partitioning $\{Q_1, Q_2, ..., Q_n\}$ of $\mathcal{Q}^{train}$, such that all $T(Q_i)$ $(= \sum_{q_j \in Q_i} |\mathbf{w}(q_j)|)$ are identical. Given an instance $\langle \mathcal{U}, n\rangle$ of the n-PARTITION problem, we construct the instance $\langle \mathcal{Q}^{train}, n\rangle$ of the n-CACHE problem as follows: for each $e \in \mathcal{U}$, insert a query q_j with $|\mathbf{w}(q_j)| = e$ into $\mathcal{Q}^{train}$. With this construction, the n-PARTITION problem is equivalent to the n-CACHE problem. Thus, the n-CACHE problem is also NP-hard. ■

As shown in the proof, our problem is NP-hard even with the simplest setting, i.e., all caches are empty.

To make matters worse, in our general problem, different servers can have different cache contents, thus they may incur different processing costs for the same query. Also, the partitioning of $\mathcal{Q}^{train}$ depends on the content of caches C_i. This renders the approximate solutions for the n-PARTITION problem unsuitable to our problem.

D. Baseline Solutions

In this subsection, we demonstrate two baseline solutions (*local frequency-based solution (**LocalF**)* and *Divgdesign-based solution* (**DIVG**)) for our problem.

Heuristic Solution (LocalF): The idea of LocalF is that: first partition a query workload $\mathcal{Q}^{train}$ into n sub-workloads $(\{Q_1, Q_2, ..., Q_n\})$ in a round-robin fashion [32], and then fill

TABLE V: Example for baseline solutions

q_1	gear, galaxy
q_2	galaxy, apple, iphone
q_3	apple, iphone, ipad
q_4	gear, iphone, apple

(a) query workload

	Server	Sub-workload Q_i	Cache C_i	Disk Seeks	Throughput
LocalF	S_1	$\{q_1, q_3\}$	{gear, galaxy}	0+3=3	$\frac{4}{\max\{3,2\}} = 1.33$
	S_2	$\{q_2, q_4\}$	{iphone, apple}	1+1=2	
DIVG	S_1	$\{q_1\}$	{gear, galaxy}	0	$\frac{4}{\max\{0,3\}} = 1.33$
	S_2	$\{q_2, q_3, q_4\}$	{apple, iphone}	1+1+1=3	
Better solution	S_1	$\{q_1, q_4\}$	{gear, iphone}	1+1=2	$\frac{4}{\max\{2,2\}} = 2$
	S_2	$\{q_2, q_3\}$	{apple, iphone}	1+1=2	

(b) comparison of different solutions

cache C_i with posting lists for most frequent terms in each sub-workload Q_i ($i \in [1, 2, ..., n]$).

Adaptation Solution [15] (DIVG): Another existing technique Divgdesign [15] in replicated databases is developed to handle SQL queries with different physical configurations. It can be converted to a solution of our problem, (termed DIVG in our work). The key procedure is that:

1) *Initialization* do the same work as LocalF;
2) *Re-assignment* re-assign queries to servers based on the lowest query costs (i.e., number of disk seeks) and update cache contents correspondingly;
3) *Iteration* repeat step 2) until cache contents reach a stable state.

Example: Table V illustrates the efficiency of the above baseline solutions in our problem. We assume that there are 2 servers (S_1 and S_2) and each server's cache capacity is 2. The query workload $\mathcal{Q}^{train}$ contains 4 queries (see Table V(a)).

For LoaclF, $\mathcal{Q}^{train}$ is divided into $Q_1 = \{q_1, q_3\}$ and $Q_2 = \{q_2, q_4\}$ in a round-robin manner. Based on frequencies of terms in Q_1 and Q_2, {gear, galaxy} and {iphone, apple} are put into C_1 and C_2. Thus, the costs on servers S_1 and S_2 are evaluated as $0 + 3 = 3$ and $1 + 1 = 2$ disk seeks respectively. The query throughput is: $4/max\{3, 2\} = 1.33$.

For simplicity, we do not illustrate the running steps for DIVG to compute C_i and Q_i. After the cache contents of C_1 and C_2 become stable, the total costs are evaluated as 0 and $1 + 1 + 1 = 3$ disk seeks. Accordingly, the query throughput is: $4/\max\{3, 0\} = 1.33$.

However, both LoaclF and DIVG are not good enough. A better solution is shown in the last row of Table V(b). In such case, the costs on servers are $1 + 1 = 2$ and $1 + 1 = 2$ disk seeks so that the query throughput is: $4/\max\{2, 2\} = 2$.

Inspired by the unsatisfied solutions, we will propose diversified caching schemes to solve our problem in a better way.

IV. Diversified Caching (DC)

Our proposed solution DC consists of two methods. At offline time, the cache admission method utilizes the training set $\mathcal{Q}^{train}$ to decide each server's cache content (cf. Sections IV-A, IV-B, IV-C). Upon receiving queries online, the online query assignment method examines them one-by-one and assigns each query to a server for execution (cf. Section IV-D). For each method, we will explore various implementation options and study their effect on the performance.

A. Framework for Offline Cache Admission

We present our framework for cache admission in Algorithm 1. The first four input parameters, namely (i) the training set $\mathcal{Q}^{train}$, (ii) the posting lists' sizes $|PL_*|$, (iii) the number of servers n, and (iv) the cache capacity $\mathcal{B}$, have been introduced in Problem 1. We employ two additional parameters α and $\#iter$. Our framework consists of a clustering phase and a merging phase.

The objective of the clustering phase is to partition $\mathcal{Q}^{train}$ into groups such that queries within the same group share as many terms as possible. We illustrate this phase in Figure 3(a). Note that it is not desirable to obtain exactly n groups in this phase. We may suffer from load balancing (during online querying) since different groups of queries may have different query costs. Thus, we suggest to obtain much more than n groups (e.g., $2^{\alpha}n$ groups) in order to give more flexibility for load balancing.

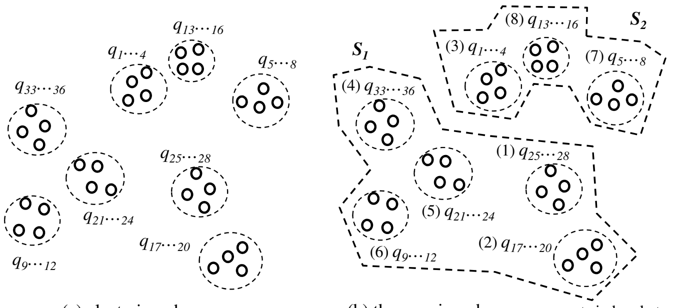
The merging phase is designed to achieve better load balancing. It would merge the above $2^{\alpha}n$ groups into n groups such that the deviation of query costs among different groups would be small. We illustrate this phase in Figure 3(b).

Algorithm 1 Diversified Caching: cache admission (offline)

```
   Input: training set $\mathcal{Q}^{train}$, posting lists' sizes $|PL_*|$, #servers $n$, cache capacity $\mathcal{B}$, parameter $\alpha$, parameter $\#iter$
1: $\{\langle Q_i \rangle_{1..2^{\alpha}n}\}$ := Clustering($\mathcal{Q}^{train}, |PL_*|, n, \mathcal{B}, \alpha, \#iter$)
   ▷ for reducing query cost
2: $\{\langle Q_i, C_i \rangle_{1..n}\}$ := Merging($\{\langle Q_i \rangle_{1..2^{\alpha}n}\}, |PL_*|, n, B, \alpha$ )
   ▷ for load balancing
3: return $\{\langle Q_i, C_i \rangle_{1..n}\}$
```

B. Offline: The Clustering Phase

Our clustering phase would invoke an existing static caching algorithm for posting lists [4], [5]. We denote this operation as `Static-Caching` ($\mathcal{Q}^{train}, |PL_*|, \mathcal{B}$), which picks a set of terms C from the training set $\mathcal{Q}^{train}$, such that the total size of the corresponding posting lists $\sum_{w \in C} |PL_w|$ is within the cache capacity $\mathcal{B}$. There are two typical implementations for `Static-Caching`.



(a) clustering phase (b) the merging phase, query costs in brackets

Fig. 3: Solution overview

- **Freq**: It selects posting lists in descending order of query frequency.
- **FreqSize**: It selects posting lists in descending order of the ratio of query frequency to posting list length.

In the following discussion, we use C to represent the set of terms selected by `Static-Caching`.

We sketch our method for the clustering phase in Algorithm 2. First, we select a set of terms C_{init} by using the total cache capacity $n\mathcal{B}$ for n servers. Then, we partition C_{init} into $2^{\alpha}n$ sets by using round-robin. Next, we run an iterative clustering procedure for $\#iter$ iterations. In each iteration, we assign each query q to a query group Q_r (Lines 5–7) and then update each server's cache content according to its query group Q_i (Lines 8–9). Finally, we return the set of query groups $\{\langle Q_i\rangle_{1..2^{\alpha}n}\}$.

As a remark, our method is similar to the well-known N-medoids clustering algorithm [27], except that we select C_i by using static caching and assign queries to servers based on their cache content.

Algorithm 2 DC: the clustering phase (offline)

```
    Input: training set $\mathcal{Q}^{train}$, posting lists' sizes $|PL_*|$, #servers $n$, cache capacity $\mathcal{B}$, parameter $\alpha$, parameter $\#iter$
1:  set $C_{init}$ := Static-Caching ( $\mathcal{Q}^{train}, |PL_*|, n\mathcal{B}$ )
2:  assign terms in $C_{init}$ to the caches $C_1, C_2, \cdots, C_{2^{\alpha}n}$ in a round-robin manner
3:  for $\#iter$ iterations do
4:      clear all $Q_i$ for $i = 1..2^{\alpha}n$
5:      for each $q \in \mathcal{Q}^{train}$ do
            ▷ implementation options for Line 6
6:          let $r \in [1..2^{\alpha}n]$ be the best cache $C_r$ for $q$
7:          insert $q$ into $Q_r$
8:      for $i$ := 1 to $2^{\alpha}n$ do
9:          $C_i$ := Static-Caching ( $Q_i, |PL_*|, \frac{\mathcal{B}}{2^{\alpha}}$ )
10: return the set of query groups $\{\langle Q_i\rangle_{1..2^{\alpha}n}\}$
```

Implementation Options:

In the above algorithm, there are different options for assigning q to the query group Q_r.

- **Miss**: It finds the group with the lowest cache misses $Miss(q, C_r) = |\mathbf{w}(q) - C_r|$, i.e., the number of disk seeks.
- **Dist**: It finds the group with the smallest Jaccard distance [1] $Dist_J(q, C_r) = 1 - \frac{|\mathbf{w}(q)\cap C_r|}{|\mathbf{w}(q)\cup C_r|}$.

In practice, q contains a few terms but C_r may contain a large number of terms. To speedup the computation of $Miss(q, C_r)$ and $Dist_J(q, C_r)$, we suggest to store C_r in a hash table so that we can compute expressions like $|\mathbf{w}(q) - C_r|$, $|\mathbf{w}(q) \cap C_r|$, $|\mathbf{w}(q) \cup C_r|$ in $O(|\mathbf{w}(q)|)$ time. This technique enables us to compute $Miss(q, C_r)$ and $Dist_J(q, C_r)$ in $O(|\mathbf{w}(q)|)$ time, regardless of the number of terms in C_r.

Example: Figure 4(a) shows the training set $\mathcal{Q}^{train}$. Suppose that there are 2 servers (S_1 and S_2), $\alpha = 1$, and each server's cache can hold 6 posting lists. In the clustering phase, we cluster similar queries (e.g., q_7 and q_8) to the same query group. Figure 4(b) shows the result after the clustering phase; there are $2^{\alpha} \cdot n = 2^1 \cdot 2 = 4$ query groups, each group can hold $6/(2^1) = 3$ posting lists.

q_1	iphone, galaxy
q_2	ipod, apple, ipad
q_3	iMac, note
q_4	iMac, gear, iphone
q_5	galaxy, gear
q_6	ipod, ipad
q_7	ipad, note, apple
q_8	ipad, apple

(a) training set $\mathcal{Q}^{train}$

i	Cache C_i	Group Q_i
1	galaxy, gear, iphone	q_1, q_5
2	ipad, ipod, apple	q_2, q_6
3	iMac, note, gear	q_3, q_4
4	apple, ipad, note	q_7, q_8

(b) after clustering: $2^{\alpha}n$ groups

Server	**Group** Q_i	**Cache** C_i
S_1	$\{q_1, q_3, q_4, q_5\}$	{gear, galaxy, iMac, iphone, note}
S_2	$\{q_2, q_6, q_7, q_8\}$	{ipad, ipod, apple, note}

(c) after merging: n groups

Fig. 4: Example for Diversified Caching, $n = 2, \alpha = 1$

Time Complexity Analysis:

Let t_{all} be the number of posting lists and t_q be the maximum number of terms for queries in $\mathcal{Q}^{train}$. The time complexity of the clustering phase (Algorithm 2) is:

$$O(\#iter \cdot 2^{\alpha}n \cdot (t_q|\mathcal{Q}^{train}| + t_{all}\log t_{all})).$$

In practice, both α and t_q are small constant values. Thus, the time complexity is linear to the number of servers n and the training set size $|\mathcal{Q}^{train}|$.

C. Offline: The Merging Phase

We present our method for the merging phase in Algorithm 3. It takes the $2^{\alpha}n$ query groups from the clustering phase as input. In each iteration (Lines 1–8), it halves the number of groups by merging groups in pairs. To achieve better load balancing, we prefer merging a small query group with a large query group, rather than merging two large query groups together. We will explore different implementation options for merging shortly. The above procedure is repeated until n query groups remain. Finally, we invoke the `Static-Caching` operation for each query group Q_i to pick a corresponding set of terms C_i for caching.

Algorithm 3 DC: the merging phase (offline)

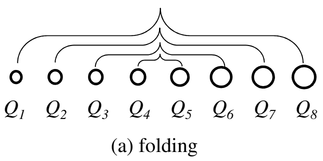
```
   Input: set of query groups $\{\langle Q_i \rangle_{1..2^\alpha n}\}$, posting lists' sizes $|PL_*|$, #servers $n$, cache capacity $\mathcal{B}$, parameter $\alpha$
1: while $\alpha > 0$ do
       ▷ implementation options for Lines 2 and 4
2:     sort $Q_i$ in ascending size and rename as $Q'_1, Q'_2, \cdots, Q'_{2^\alpha n}$
3:     for $i := 1$ to $2^{\alpha-1} n$ do
4:         find two remaining groups for merging, say $Q'_j$ and $Q'_k$
5:         merge $Q'_j$ and $Q'_k$ into a group $Q_{(i)}$
6:         remove the groups $Q'_j$ and $Q'_k$
7:     rename each group $Q_{(i)}$ to $Q_i$ for $i = 1..2^{\alpha-1}$
8:     $\alpha := \alpha - 1$
9: for each $i$ from 1 to $n$ do
10:    $C_i :=$ Static-Caching ( $Q_i, |PL_*|, \mathcal{B}$ )
11: return the set of pairs $\{\langle Q_i, C_i \rangle_{1..n}\}$
```

Implementation Options:

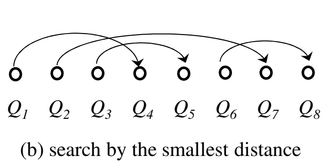
Intuitively, it is beneficial to group queries into clusters, such that queries within the same cluster tend to access similar terms. If a server caches the frequent terms of this cluster, the hit ratio of this server to process queries can be improved.

In the above algorithm, there are different options for pairing the groups for merging.

- **Fold-by-#query**: First, it sorts the groups in ascending order of the number of queries $|Q_i|$, as shown in Figure 5(a). Then, it iteratively merges the i-th smallest group with the i-th largest group. For example, in Figure 5(a), we merge Q_1 and Q_8, merge Q_2 and Q_7, merge Q_3 and Q_6, and merge Q_4 and Q_5.
- **Fold-by-#term**: This option is similar to fold-by-#query, except that it sorts the groups in ascending order of the number of distinct terms in Q_i, i.e., $|\bigcup_{q \in Q_i} \mathbf{w}(q)|$.
- **Search-by-distance**: It merges each group with the closest remaining group in terms of the Jaccard distance $Dist_J(q, C_i)$. We depict this option in Figure 5(b).
- **Search-by-union**: This option is similar to search-by-distance, except that it merges each group with the remaining group such that the merged group contains the least number of distinct terms.



(a) folding



(b) search by the smallest distance

Fig. 5: Implementation options for merging

Continuing with the previous example in Figure 4(b), after the clustering phase, we would need to merge those $2^\alpha n$ query groups to n query groups. Suppose that we merge groups by using **search-by-distance**. The Jaccard distance of closest pairs among these groups are $Dist_J(Q_1, Q_3) = 1 - \frac{1}{5} = \frac{4}{5}$ and $Dist_J(Q_2, Q_4) = 1 - \frac{2}{4} = \frac{1}{2}$. Figure 4(c) shows the n result groups after merging query groups.

Time Complexity Analysis:

Let t_{all} be the number of posting lists and t_q be the maximum number of terms for queries in $\mathcal{Q}^{train}$. Lines 9–10 take $O(t_q|\mathcal{Q}^{train}| + t_{all} \log t_{all})$ time.

In addition, the time complexity of merging (Lines 1–8) depends on the implementation option discussed above. It takes $O((2^\alpha n) \log(2^\alpha n))$ for 'fold-by' options, but $O((2^\alpha n)^2)$ for 'search-by' options. Since α is a small constant in practice, it is feasible to run the merging phase.

D. Online Query Assignment

We proceed to investigate how the broker assigns queries to servers during online time. In order to maximize the query throughput, the broker should (i) assign each query to a server with low query cost, and (ii) balance the load among different servers.

We propose our online query assignment method for the broker in Algorithm 4. To facilitate load balancing, the broker employs a load counter $load_i$ for each server S_i to accumulate its total estimated query processing cost so far. In order to estimate the query cost on servers, the broker keeps the information of cached terms $C_1, C_2, \cdots, C_n$ and posting lists' sizes $|PL_*|$ in hash tables. This requires only $O(nt_{all})$ memory space, where t_{all} is the number of posting lists.

The algorithm processes queries in the incoming stream $\mathcal{Q}^{test}$ one-by-one. For the current query q, the algorithm first estimates the query cost of q and then picks the server with the smallest estimated query cost. Then, it refines the server choice according to the loads of servers. Finally, it forwards q to the chosen server S_r for processing, and then increases the load counter of S_r by the estimated query cost.

Algorithm 4 DC: query assignment (online)

```
   Input: a stream of queries $\mathcal{Q}^{test}$
   Internal values: #servers $n$, posting lists' sizes $|PL_*|$, the cached terms and the load of each server $S_i$ $\{\langle C_i, load_i \rangle_{1..n}\}$
1: for each $q$ in $\mathcal{Q}^{test}$ do
2:     for each $i$ from 1 to $n$ do
3:         estimate Cost $(q, C_i)$ by using $C_i, |PL_*|$
4:         refine the server choice according to $load_i$
5:     forward $q$ to the chosen server $S_r$ for processing
6:     $load_r \leftarrow load_r +$ Cost $(q, S_r)$
```

Estimating the query cost:

We focus on estimating the I/O cost for query q on server S_i.

A simple way is to estimate the query cost on S_i by using the number of cache misses on C_i:

$$Miss(q, C_i) = |\mathbf{w}(q) - C_i|. \quad (3)$$

because each cache miss incurs a disk seek. As mentioned before, the function $Miss(q, C_i)$ takes only $|\mathbf{w}(q)|$ computation time.

Nevertheless, the above equation may not be accurate. A more accurate equation would consider the number of disk pages accessed by random read and sequential read. When a server reads a posting list from the disk, it retrieves the first disk page by random read and the subsequent disk pages by sequential read. Thus, we obtain the following cost equation:

$$DiskCost(q, C_i) = \sum_{w \in \mathbf{W}(q) - C_i} 1 + \texttt{round}(\phi \cdot \frac{|PL_w|}{d}) \quad (4)$$

where d is the number of posting list entries per disk page, and ϕ is the ratio of sequential disk read time to random disk read time. The typical value of ϕ is $\frac{1}{100}$ [36].

Load balancing: We propose two load balancing heuristics: tie-based heuristic and score-based heuristic.

Since the above cost equations return integer values, it is possible to have ties, i.e., more than one servers have the lowest query cost for q. The tie-based heuristic detects such scenario and chooses the server with the smallest load $load_i$ among the ties.

The score-based heuristic combines both the query cost $Cost(q, C_i)$ and the load $load_i$ of server S_i into the following equation:

$$Score(q, S_i) = \frac{Cost(q, C_i)}{\max_{i=1}^{n} Cost(q, C_i)} - \frac{1}{\delta} \cdot \left(1 - \frac{load_i}{\max_{i=1}^{n} load_i}\right)$$

where the parameter δ determines the weight of load balancing. The broker will choose the server with the smallest $Score(q, S_i)$ for q. According to this equation, a server with smaller $Cost(q, C_i)$ or smaller $load_i$ is more likely to be chosen. The term $(1 - \frac{load_i}{\max_{i=1}^{n} load_i})$ represents the imbalance ratio of S_i compared to the highest loaded server. In our experiments, the typical value of $(1 - \frac{load_i}{\max_{i=1}^{n} load_i})$ ranges from 5% to 30%. Thus, we recommend setting δ between 5–30% for normalization.

Heuristics for online query assignment: As a result, we have four combinations for online query assignment.

- Miss and tie-based balancing (M+T)
- Miss and score-based balancing (M+S)
- DiskCost and tie-based balancing (D+T)
- DiskCost and score-based balancing (D+S)

We will examine the effectiveness of the above heuristics for online query assignment in Section V.

V. Experiments

In this section, we experimentally evaluate the performance of our proposed solution and competitors on a real dataset. Section V-A introduces the experimental setting. Section V-B explores the effectiveness of caching policies on the solutions. Section V-C examines the effectiveness of various options in our proposed solution. Section V-D compares the performance of the solutions with respect to various parameters.

TABLE VI: Statistics of data and queries

Query Log	Statistics
Total data size	76 GB
Inverted index size	6.5 GB
Number of queries	1,000,000
Number of distinct queries	495,073

A. Experimental settings

All the experiments are conducted on a cluster of 9 commodity machines. We use 1 machine as the query broker and the rest machines as servers. Each machine has a Intel i5-3570 3.4GHz processor and a 1TB SATA disk. We deploy a Lucene[2] based search implementation in each machine.so

We use the ClueWeb web data[3], which is a real collection of web pages crawled in 2009. Also, we use the AOL query log[4], which is a real collection of queries submitted to the AOL search engine. To ensure replaying queries on Lucene in a manageable time, we draw a random sample of 76 GB data as the dataset and 1 million queries as the query workload. Then, we replicate this dataset and its inverted index on each server. Table VI summaries the characteristics of our dataset and query set.

For other experimental settings, we follow the typical options in the literature. We divide the real query log into two halves, as setting in [26], [36]: (i) the first half (500,000 queries) as the training set $\mathcal{Q}^{train}$ (for building cache content), and (ii) the second half (500,000 queries) as the testing set $\mathcal{Q}^{test}$ (for the replay). The page (block) size in the system is 4KB [17], [33]. Table VII shows our parameter settings.

In the following experiments, we focus on the efficiency of online query processing rather than offline pre-processing. For search engines, the online query throughput and query response time are the most important measurement for efficiency, whereas the offline pre-processing time is not observed by users. Besides, we measure (i) the query throughput (in number of queries answered per second) as

$$\text{throughput} = \frac{|\mathcal{Q}^{test}|}{\max_{i=1}^{n} T_{C_i}(Q_i)}$$

and (ii) the imbalance ratio as

$$\text{imbalance ratio} = \left(1 - \frac{\min_{i=1}^{n} T_{C_{i'}}(Q_{i'})}{\max_{i=1}^{n} T_{C_i}(Q_i)}\right) \times 100\%$$

Regarding these two metrics, the ideal solution should achieve high throughput and low imbalance ratio.

Methods for Comparison:

We call our solution as **DC** and name its variants in later subsections. We adapt existing methods in the literature as competitor methods (cf. Table VIII).

[2]http://lucene.apache.org

[3]http://lemurproject.org/clueweb09

[4]http://www.gregsadetsky.com/aol-data

TABLE VII: Parameter settings

Parameter	Value range	Default
Number of servers n	[2, 8]	8
Cache size (GB)	[0.5, 2]	1
Load-balancing factor α	[0, 5]	2
the number of iterations	[1, 100]	10

TABLE VIII: Competitor methods

Method	Offline	Online
Uniform	static caching on the global training set	round-robin
LocalF	static caching on local training sets	round-robin
DIVG	[15]	lowest miss, tie-breaking

Uniform: In the offline phase, it fills all caches with same content (cf. Section I);

LocalF: In the offline phase, it partitions queries in a round-robin manner and then fills each cache according to its assigned query workload (stated in Section III-D);

DIVG: In the offline phase, this method adapts the divergent-design-based solution (stated in Section III-D). In the online phase, the broker assigns each query to a server with the lowest misses (i.e., number of disk seeks).

B. *Effect of Static Caching Policies*

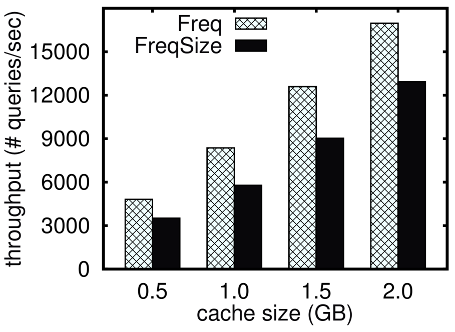
We first evaluate the effect of static caching policies used in the methods. As stated in Section III, each server employs a posting list cache and uses a static caching policy. We test with two classic static caching policies [4], [5]: (i) *Freq*, which fills the cache by posting lists with high access frequency, and (ii) *FreqSize*, which fills the cache by posting lists with high frequency to posting list length ratio.

Figure 6 shows the performance of Uniform under different static caching policies. Freq achieves a better throughput than FreqSize, although FreqSize has a higher hit ratio than Freq. Since FreqSize favors caching shorter posting lists than Freq, a cache hit in FreqSize would save less I/O cost when compared to Freq. We have also repeated this experiment for other methods (LocalF, DIVG, DC) and obtained similar trends. Therefore, we set *Freq* as default static caching policy in the remaining experiments.

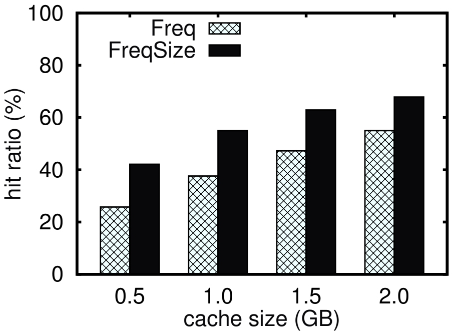
C. *Effect of Options and Parameters on Diversified Caching*

We proceed to investigate the effect of various options (e.g., clustering options, merging options, query assignment options) and parameters (e.g., $\alpha, \#iter$) on the performance of our proposed Diversified Caching solution.

Figure 7 plots the performance of Diversified Caching with respect to different combinations of clustering and merging options. In this experiment, we fix the online query assignment option to: MISS and tie-based balancing. The labels on the x-axis indicate options in the clustering phase (cf. Section IV-B), e.g., MISS, DIST. The option NO refers to the case of disabling the iterative clustering procedure (for-loop). Within each group of bars, the bars are shown according to their options in the merging phase (cf. Section IV-C). We represent

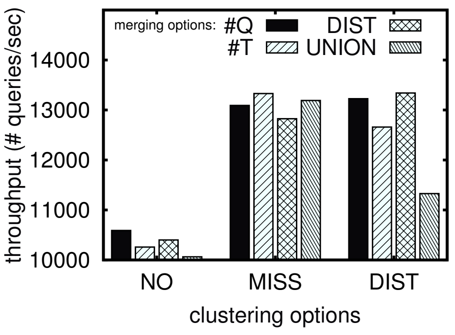


(a) throughput

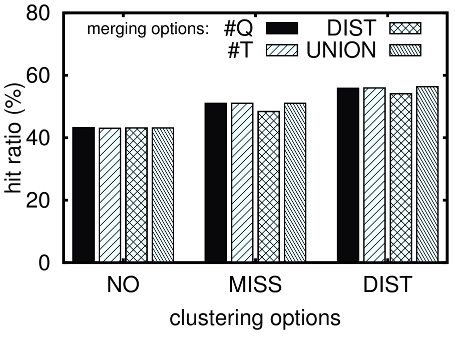


(b) hit ratio

Fig. 6: Effectiveness of static caching policy on Uniform



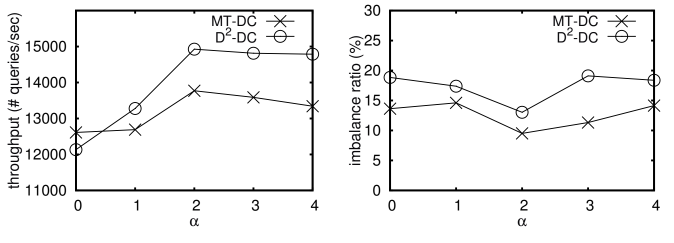
(a) throughput



(b) hit ratio

Fig. 7: Effectiveness of clustering and merging options on our solution

the options fold-by-#query, fold-by-#term, search-by-distance, search-by-union by using #Q, #T, DIST, UNION, respectively. Clearly, it is not desirable to disable the iterative clustering procedure; the bars in the group NO cannot achieve high throughput. Within the group MISS, the best merging option is #T. Within the group DIST, the best merging option is DIST. We denote these two methods as MT-DC (meaning MISS+#T) and D^2-DC (meaning DIST+DIST) and then use them in the subsequent experiments.

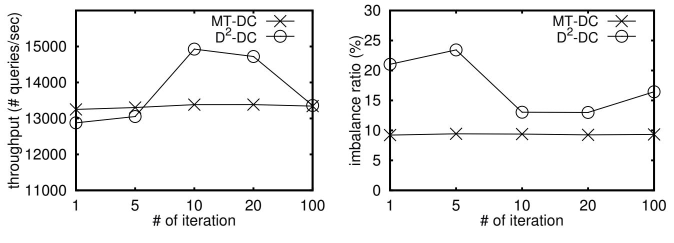


(a) throughput (b) imbalance ratio

Fig. 8: Performance vs. α

Figure 8 shows the performance of our methods as a function of α. The performance remains stable across a range of α values (from 2 to 4). The best performance is obtained when $\alpha = 2$. Thus, we fix $\alpha = 2$ in subsequent experiments.

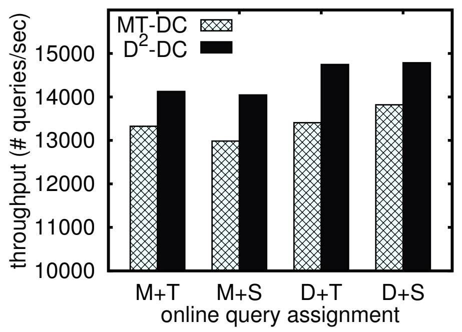
Figure 9 plots the performance of our methods with respect to the number of iterations. The throughput of MT-DC is insensitive to the number of iterations. D^2-DC performs better than MT-DC when the number of iterations is sufficiently high. In the remaining experiments, we set the number of iterations to 10.



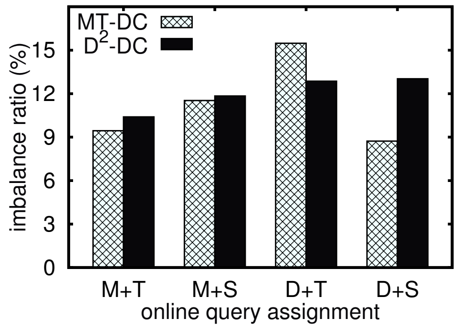
(a) throughput (b) imbalance ratio

Fig. 9: Performance vs. # iterations

Next, we examine the effect of online query assignment options on our methods MT-DC and D^2-DC. Each combination is labeled with two letters, where the first letter denotes the query cost equation (M for Miss, D for DiskCost) and the second letter denotes the load balancing option (T for tie-based, S for score-based). For score-based balancing (S), we set the parameter δ to 0.05. A more accurate cost equation (D) helps improving the throughput. Thus, D+T and D+S perform better than M+S and M+T, respectively. Since D+S achieves the best throughput, we use this setting in the following experiments.



(a) throughput



(b) imbalance ratio

Fig. 10: Effectiveness of online query assignment options on our solution

D. Performance Comparisons

We compare the performance of our proposed methods (MT-DC and D^2-DC) with competitors (Uniform, LocalF, and DIVG).

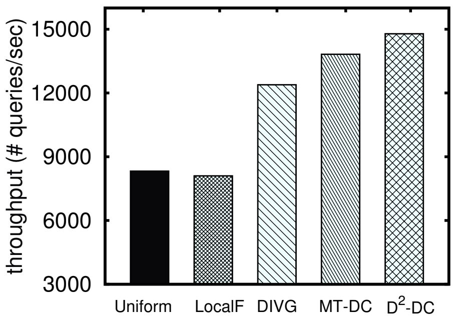
First, we plot the experimental results in Figure 11, while fixing the parameters to their default values. Our methods achieve better throughput than DIVG and outperform baseline solutions (Uniform and LocalF) significantly, as shown in Figure 11(a). In terms of the imbalance ratio, our methods perform much better than DIVG (cf. Figure 11(b)).

Since LocalF and Uniform have similar performance, we ignore Uniform in the following experiments.

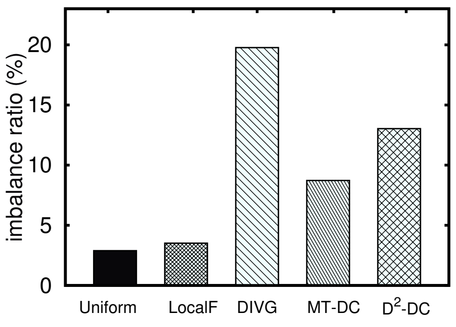
We proceed to test the scalability of the methods with respect to the cache size, the number of servers, and the data size.

Effect of cache size: First, we test the performance of the methods by varying the cache size (of each server) from 0.5 GB to 2.0 GB. Figures 12(a)(b)(c) show the query throughput, the cache hit ratio, and the imbalance ratio of the methods. In general, the query throughput of all methods rises with the cache size. Our methods (MT-DC and D^2-DC) achieve better throughput than the competitors (DIVG and LocalF), and their performance gap widens as the cache size increases.

Effect of number of servers: Next, we examine the performance of the methods with respect to the number of servers n (from 2 to 8). As illustrated in Figure 13, the throughput



(a) throughput



(b) imbalance ratio

Fig. 11: Overall performance comparison

of each method increases proportionally with n. When n increases, the total cache capacity (of n servers) increases, thus the hit ratio also increases. On the other hand, it becomes more difficult to balance the load across servers when n is large.

Effect of data size: Finally, we evaluate the scalability of our methods by varying data size from 15GB to 76GB.

We obtain smaller datasets by sampling from the default 76 GB dataset. For each dataset, we set its cache size proportionally based on its data size. Figures 14(a)(b)(c) show that our proposed methods achieve higher throughput, higher hit ratio, and lower imbalance ratio than competitors.

VI. Conclusion and Future Work

In this paper, we proposed a novel caching problem for parallel search engines that employ replication on inverted index. We showed that the problem is computationally hard, and hence designed a diversified caching framework with various implementation options. Experimental results demonstrated that our methods achieved high throughput and low imbalance ratio.

As future work, we will focus on studying the diversified caching problem for other types of caching policies and caches (e.g., document cache [34], snippet cache [14]) in document servers.

Acknowledgments The research is supported by grant GRF 152201/14E from Hong Kong RGC.

References

[1] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.

[2] C. S. Badue, R. A. Barbosa, P. B. Golgher, B. A. Ribeiro-Neto, and N. Ziviani. Basic issues on the processing of web queries. In *SIGIR*, pages 577–578, 2005.

[3] R. A. Baeza-Yates, C. Castillo, F. Junqueira, V. Plachouras, and F. Silvestri. Challenges on distributed web retrieval. In *ICDE*, pages 6–20, 2007.

[4] R. A. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. The impact of caching on search engines. In *SIGIR*, pages 183–190, 2007.

[5] R. A. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. Design trade-offs for search engine caching. *TWEB*, 2(4), 2008.

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

[7] L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[8] P. A. Bernstein, I. Cseri, N. Dani, N. Ellis, A. Kalhan, G. Kakivaya, D. B. Lomet, R. Manne, L. Novik, and T. Talius. Adapting microsoft sql server for cloud computing. In *ICDE*, pages 1255–1263, 2011.

[9] D. Broccolo, C. Macdonald, S. Orlando, I. Ounis, R. Perego, F. Silvestri, and N. Tonellotto. Load-sensitive selective pruning for distributed search. In *CIKM*, pages 379–388, 2013.

[10] F. Cacheda, V. Carneiro, V. Plachouras, and I. Ounis. Performance comparison of clustered and replicated information retrieval systems. In *ECIR*, pages 124–135, 2007.

[11] F. Cacheda, V. Plachouras, and I. Ounis. A case study of distributed information retrieval architectures to index one terabyte of text. *Inf. Process. Manage.*, 41(5):1141–1161, 2005.

[12] B. B. Cambazoglu, A. Catal, and C. Aykanat. Effect of inverted index partitioning schemes on performance of query processing in parallel text retrieval systems. In *ISCIS*, pages 717–725, 2006.

[13] B. B. Cambazoglu, E. Kayaaslan, S. Jonassen, and C. Aykanat. A term-based inverted index partitioning model for efficient distributed query processing. *TWEB*, 7(3):15, 2013.

[14] D. Ceccarelli, C. Lucchese, S. Orlando, R. Perego, and F. Silvestri. Caching query-biased snippets for efficient retrieval. In *EDBT*, pages 93–104, 2011.

[15] M. P. Consens, K. Ioannidou, J. LeFevre, and N. Polyzotis. Divergent physical design tuning for replicated databases. In *SIGMOD Conference*, pages 49–60, 2012.

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[17] B. K. Debnath, S. Sengupta, and J. Li. Flashstore: High throughput persistent key-value store. *PVLDB*, 3(2):1414–1425, 2010.

[18] Q. Gan and T. Suel. Improved techniques for result caching in web search engines. In *WWW*, pages 431–440, 2009.

[19] B.-S. Jeong and E. Omiecinski. Inverted file partitioning schemes in multiple disk systems. *IEEE Trans. Parallel Distrib. Syst.*, 6(2):142–153, 1995.

[20] X. Long and T. Suel. Three-level caching for efficient query processing in large web search engines. *World Wide Web*, 9(4):369–395, 2006.

[21] R. Ma. Baidu distributed database. In *SACC*, pages 426–435, 2010.

[22] M. Marín, V. G. Costa, and C. Gómez-Pantoja. New caching techniques for web search engines. In *HPDC*, pages 215–226, 2010.

[23] A. Moffat, W. Webber, and J. Zobel. Load balancing for term-distributed parallel retrieval. In *SIGIR*, pages 348–355, 2006.

[24] S. B. Navathe and M. Ra. Vertical partitioning for database design: A graphical algorithm. In *SIGMOD Conference*, pages 440–450, 1989.

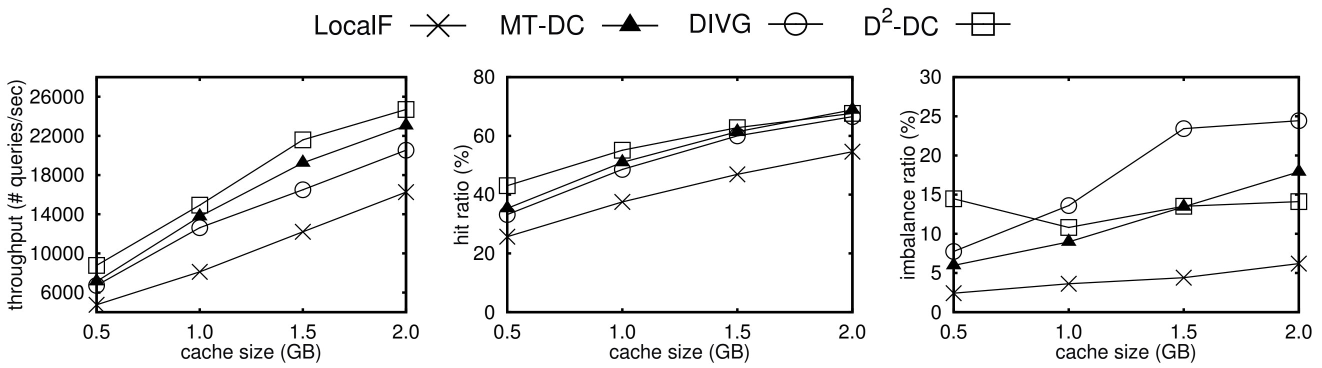
[25] R. Ozcan, I. S. Altingövde, and Ö. Ulusoy. Static query result caching revisited. In *WWW*, pages 1169–1170, 2008.

[26] R. Ozcan, I. S. Altingövde, and Ö. Ulusoy. Cost-aware strategies for query result caching in web search engines. *TWEB*, 5(2):9, 2011.

[27] H.-S. Park and C.-H. Jun. A simple and fast algorithm for k-medoids clustering. *Expert Syst. Appl.*, 36(2):3336–3341, 2009.

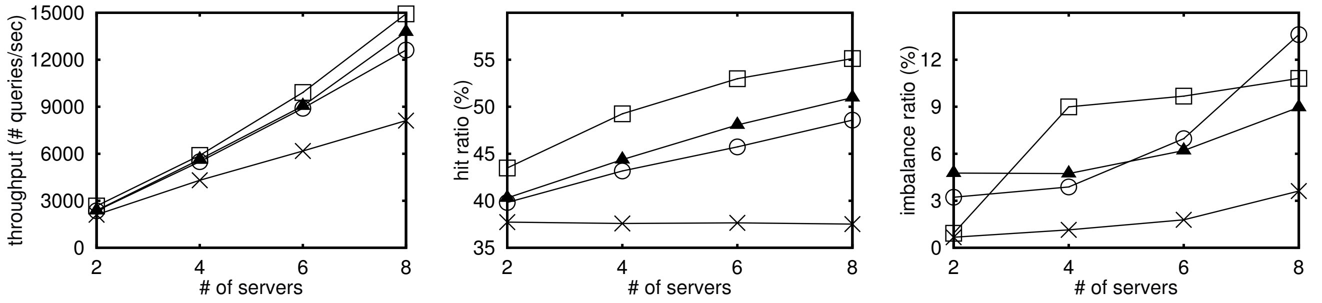
[28] D. Puppin, F. Silvestri, and D. Laforenza. Query-driven document partitioning and collection selection. In *Infoscale*, page 34, 2006.

[29] D. Puppin, F. Silvestri, R. Perego, and R. A. Baeza-Yates. Load-balancing and caching for collection selection architectures. In *Infoscale*, page 2, 2007.



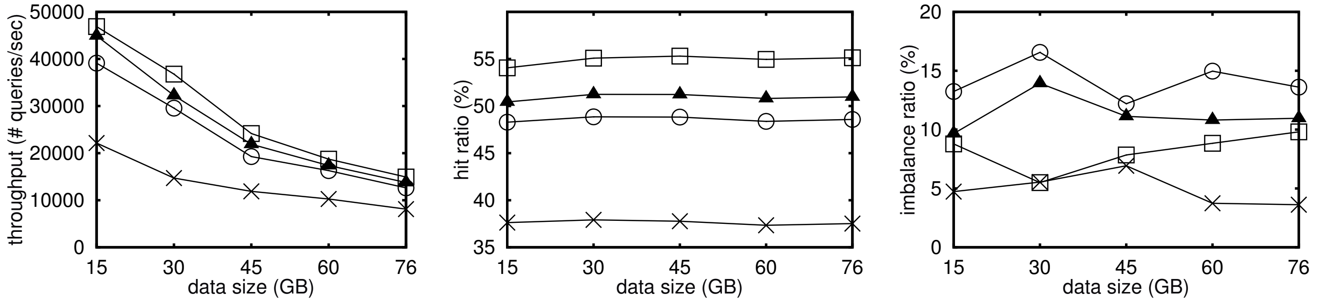
(a) throughput (b) hit ratio (c) imbalance ratio

Fig. 12: Performance vs. cache size



(a) throughput (b) hit ratio (c) imbalance ratio

Fig. 13: Performance vs. number of servers



(a) throughput (b) hit ratio (c) imbalance ratio

Fig. 14: Performance vs. data size

[30] Z. Shang and J. X. Yu. Catch the wind: Graph workload balancing on cloud. In *ICDE*, pages 553–564, 2013.

[31] A. Y. Teymorian, O. Frieder, and M. A. Maloof. Rank-energy selective query forwarding for distributed search systems. In *CIKM*, pages 389–398, 2013.

[32] A. Tomasic and H. Garcia-Molina. Performance of inverted indices in distributed text document retrieval systems. In *PDIS*, pages 8–17, 1993.

[33] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query processing techniques for solid state drives. In *SIGMOD Conference*, pages 59–72, 2009.

[34] A. Turpin, Y. Tsegay, D. Hawking, and H. E. Williams. Fast generation of result snippets in web search. In *SIGIR*, pages 127–134, 2007.

[35] A. P. U. Manber and J. Robison. Yahoo! In *Communications of the ACM*, volume 43, pages 124–135, 2000.

[36] 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. In *SIGIR*, pages 693–702, 2013.

[37] J. Xu and W. B. Croft. Cluster-based language models for distributed retrieval. In *SIGIR*, pages 254–261. ACM, 1999.

[38] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.