ARTICLE IN PRESS

Future Generation Computer Systems ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

EDS: An Efficient Data Selection policy for search engine storage architectures

Xinhua Dong [a], Ruixuan Li [a,*], Heng He [a], Xiwu Gu [a], Mudar Sarem [b], Meikang Qiu [c], Keqin Li [d]

[a] School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, Hubei, 430074, China
[b] School of Software Engineering, Huazhong University of Science and Technology, Wuhan, Hubei, 430074, China
[c] Department of Computer Science, Pace University, New York, NY 10038, USA
[d] Department of Computer Science, State University of New York, New York, NY 12561, USA

HIGHLIGHTS

- We introduce, define and analyze the knapsack problem in different storage architectures.
- We find some critical factors via derivation and comparison.
- We present an Efficient Data Selection (EDS) policy for search engine cache management.
- We carry out a series of experiments to study essential factors of the data selection.
- Extensive experiments show that the EDS policy using static cache further improves the performance of search engines.

ARTICLE INFO

Article history:
Received 15 September 2015
Received in revised form
1 February 2016
Accepted 23 February 2016
Available online xxxx

Keywords:
Search engine
Data selection
Solid state disk
Hybrid storage architecture
Cache

ABSTRACT

Caching is an effective optimization in search engine storage architectures. Many caching algorithms have been proposed to improve retrieval performance. The data selection policy of search engine cache management plays an important role, which carefully places the data in memory or other storage, such as solid state disks (SSDs). Considering that the historical query log has a guiding role for the future query, we present an Efficient Data Selection (EDS) policy for search engine cache management, which views cache media as a knapsack, and views results and posting lists as items. The best benefit of EDS can be computed by greedy algorithms. We carry out a series of experiments to study the essential factors of the data selection in different architectures, including hard disk drive (HDD), SSD, and SSD-based hybrid storage architectures. The hybrid storage architecture is a two-level cache architecture, which uses SSD as a secondary cache for the memory. Our main goal is to improve the performance of the search engines and reduce the cost of the servers on two-level cache architecture. The experimental results demonstrate that our proposed policy improves the hit ratio by 20.04% as well as the retrieval performance on HDD, SSD, and hybrid architecture by 31.98%, 28.72% and 23.24%, respectively.

1. Introduction

With the development of computer technology in hardware, the low I/O performance of hard disk drive (HDD) becomes the major bottleneck in modern large-scale search engines. Fortunately, the emerging solid state disk (SSD) technology brings new and promising opportunities to I/O-intensive applications. Unlike traditional rotating media, the SSD is based on semiconductor chips which provide many desired technical merits, such as low power consumption [1], compact size, shock resistance, and most importantly, ultra-high performance for random data access. For example, the random reads in the SSD are one to two orders of magnitude faster than in the HDD [2]. As a result, the SSD has been employed in many industrial settings, including Facebook [3], Microsoft Azure [4], Google [5] and Baidu [6] which has already used SSD to completely replace the HDD in its infrastructure.

However, two potential issues may complicate the full adoption of the SSD in large-scale search engines. First, the current average

* Corresponding author.
E-mail addresses: xhdong@hust.edu.cn (X. Dong), rxli@hust.edu.cn (R. Li), heheng@hust.edu.cn (H. He), guxiwu@hust.edu.cn (X. Gu), mudar66@hotmail.com (M. Sarem), mqiu@pace.edu (M. Qiu), lik@newpaltz.edu (K. Li).

http://dx.doi.org/10.1016/j.future.2016.02.014
0167-739X/

Please cite this article in press as: X. Dong, et al., EDS: An Efficient Data Selection policy for search engine storage architectures, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.02.014

cost per GB of the SSD is 7 times more than that of the HDD [7]. In addition, the existing data in large-scale search engines is extraordinarily large, and it is not suitable to store all the data on the SSD. For example, Google makes the use of a large number of cheap servers with the HDD to provide high quality services [8]. Second, since the SSD has a limit on block erasure count, the combination of more stressful workload and fewer available erasure cycles reduce the useful lifetime of the SSD, in some cases, to less than one year [9]. Without any major change to the existing HDD-based storage systems used in search engines, our team has proposed an SSD-based hybrid storage architecture with memory as the first-level cache and an SSD as the second-level cache [10].

In a modern search engine, caching is the preferred technique for attaining performance. Two types of cached data (namely results and inverted lists) are dominant. There are some differences between caching the results and the inverted lists. First, the result entries are small and similar in size, while the inverted list entries are usually large and variable in size [10]. In addition, only parts of the inverted lists are required during computing. Second, the results are prominently relevant to the queries and time-sensitive, while the inverted lists are relatively stable. Over the past years, many caching techniques have been developed and used in search engines. In order to reduce the query response time, the search engines commonly dedicated portions of the servers' memory to cache certain query results [11–14], posting list [15–17], intersection [18], document [19], score, and snippet [20]. Caching means copying frequently or recently accessed parts of the data from high-capacity but slow storage devices (HDD) to low-capacity but fast storage devices (Memory or SDD). The data selection means selecting the effective data placed in the memory or the SSD. These caches avoid excessive disk access and repeated computation. As the traditional evaluation indexes, the hit ratio and the average query latency are considered in the performance evaluation of our experiments.

Due to the wide speed gap between the random read and the sequential read in the HDD, the benefit of the cache hit has been largely attributed to the saving of the expensive random read operations. Therefore, the early studies focused on the improvement of the hit ratio in HDD-based search engine architecture, and put forward some classic policies [16,17], such as Freq, FreqSize etc. In an SSD-based search engine infrastructure, now the benefit of the cache hit should impute to the saving of the random read and the subsequent reads. Since SSD is replacing HDD, the cache hit ratio is no longer a reliable reflection of the actual query latency because a larger data items being found in the cache yields a higher query latency improvement over a smaller data items [2]. The frequency was found to be as a core factor in the SSD experiments. In an SSD-based hybrid search engine infrastructure, our previous proposed CBSLRU [10] algorithm gained higher hit ratios than the traditional LRU.

However, by empirical value, the frequency and the size are the major consideration factors in some related work. Due to the lack of theoretical guidance, it is difficult to find other factors that affect the performance in the related literature. So, the applications may not work well in different hardware environments. We assume that the historical query log has a guiding role for the future query [12]. Then, a set of query features could be found through query log. When the user submits a query, the data selection policy is tuned to use these features. In order to improve the efficiency of the data selection, we have introduced the knapsack problem, which views cache media as knapsack and views result and posting list as items, and then we have used a greedy algorithm to calculate the retrieval time. Thus, our proposed EDS value is not only associated with the frequency and the size, but also related to the hardware and the software parameters, such as the seek time and the rotational latency of the HDD, the time of obtaining a block from the HDD, etc. The EDS can be better applied to different storage architectures. The extension of EDS can also improve the response time in the HDFS, and enhance the real-time ability of the retrieval.

We have made three main contributions in our work. First, we have described and analyzed the knapsack problem in different storage architectures. Second, through derivation and comparison, we have found some critical factors which affect the performance of the search engines. Third, we have proposed EDS policy which places the efficient data to be cached either in the memory or in the SSD.

The rest of the paper is organized as follows. In Section 2, we introduce the background of the caching techniques and the storage architectures of the search engines, and discuss the related work. Section 3 presents different storage architectures for search engines. In Section 4, we analyze the knapsack problem in different storage architecture. In Section 5, we perform the derivation of the EDS in different storage architectures. Section 6 shows the results of the performance evaluation. Finally, in Section 7, we conclude this study and discuss the future work.

2. Background and related work

In the following subsections, we will introduce the caching techniques used in search engines, the storage architecture of a search engine, and the knapsack problem.

2.1. Caching techniques in search engines

In general, search engine caches can be classified into two categories: static caches [15] and dynamic caches [21]. The static caches try to capture the access locality of the data items. Past data access logs are utilized to determine the data that should be cached. Typically, the items that are more frequently accessed in the past are preferred over the infrequently accessed items for caching. The static caches need to be periodically updated, depending on the variation of the access frequencies of the items. On the other hand, the dynamic caches try to capture the recency of the data access. The data that is more likely to be accessed in the near future remains in the cache. The challenge reported in the previous research was to develop cache eviction policies [13,22]. Recently, the current challenge is to devise effective policies to keep the inverted lists and the results used search terms in cache [23].

Caching is an effective optimization in search engine. The caching techniques can be classified into Two-Level caching and Multi-Level caching in search engines. Saraiva et al. [24] evaluated a two-level caching architecture using result and list caching on the search engine TodoBR. The result caching filtered out the repetition in the query stream by caching the complete results of the previous queries for a limited time window [21,13]. At the lower level inside each index server, the list caching was used at a lower level in each participating machine to keep the inverted lists of the frequently used search terms in main memory. Besides the result and the list caching, adding other cached object can be considered as a Multi-Level caching. Long and Suel [25] proposed and evaluated a three-level caching scheme that added an intermediate level of caching. On this basis, Wang et al. [2] also carried out a series of experiments on documents and snippets in the proposed web search engine architecture. Ozcan et al. [18] suggested a five-level static cache architecture for web search engines.

2.2. Storage architecture of search engine

In one-level architecture, the cache is a memory region adopted to store the most frequently used data. Normally, most of the data is stored in secondary storage medium such as HDD or

Please cite this article in press as: X. Dong, et al., EDS: An Efficient Data Selection policy for search engine storage architectures, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.02.014

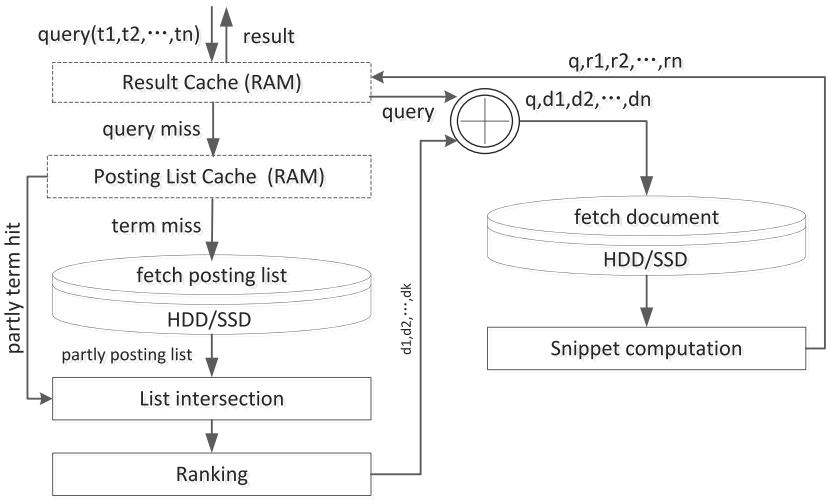


Fig. 1. One-level search engine architecture.

SSD. However, there are some search engines such as Baidu which has already used SSD to completely replace HDD in their infrastructures [6]. That is to say, there are two situations in one-level architecture, one can be called "Memory + HDD" and the other is called "Memory + SSD".

Considering the special I/O performance of the SSD, the researches have been attracted on such SSD-based hybrid storage architecture. With its excellent random read performance, the SSD can work well as a read cache in front of a larger HDD [26,27]. G. Soundararajan et al. [9] proposed Griffin, a hybrid storage device using the HDD as a write cache for an SSD. Suk et al. [28] suggested a hybrid file system, called hybridFS, where the primary objective of which is to put together attractive features of the HDD and the SSD devices, to construct a large-scale virtualized address space with a minimum cost. Jaechun No [29] proposed a hybrid file system using N-hybridintegrating SSD and HDD devices in a cost-effective manner to build a large-scale virtual storage capacity. Kai et al. [30] introduced a Hybrid SSD approach that combined DRAM, Phase Changed Memory (PCM), and Flash. The performance gap between the HDD and the DRAM was narrowed by using Hybrid SSD more than using the pure Flash SSD while delivering a longer endurance life.

2.3. Knapsack problem

The knapsack problem is a problem in combinatorial optimization. Given a set of items, each with a weight and a value, determine the number of items to be included in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. Under the condition of memory capacity restriction, the results and the posting lists to be cached are as many as possible in search engine cache management. The scenario of our study is similar to the knapsack problem. Therefore, we use knapsack problem to optimize the choice of the cache. The most common problem being solved is the 0–1 knapsack problem, which restricts the number x_i of copies of each kind of item to zero or one. Knapsack problem has been developed and used in search applications [31,32]. Huang and Xia [33] modeled a knapsack problem based on the bandwidth and the capacity for a search engine to allocate the inverted index in the flash memory. And they also proposed the standard greedy algorithm to approximate the optimal solution. However, they aimed to maximize the amount of inverted indexes which had been allocated in the flash memory, but they lacked theoretical derivation for the model. Chan et al. [34] proposed a cache replacement policy called Location Dependent Cooperative Caching (LDCC) to improve the performance of the LDISs and presented a 0–1 knapsack problem and an approximate algorithm to solve the cache replacement problem. Also, they advised a comprehensive cost functions in cache replacement and integrating factors like cache size, access frequency, energy consumption, and the validity probability obtained.

3. Storage architectures of search engines

In this section, we briefly present the storage architecture of a search engine, including one-level and two-level architectures. And then, we give the data management process of the two-level search engine architecture.

3.1. One-level architecture

The cache storage structure can be divided into two levels: level 1 cache (L1 cache) and level 2 cache (L2 cache). L1 cache refers to the memory, and its capacity is usually several GB to dozens of GB. While, L2 cache refers to the SSD, and its capacity is usually dozens of GB to hundreds of GB. The normal search engine employs one-level architecture (Fig. 1). Upon receiving a user's query q with n terms $t_1, t_2, \ldots, t_n$, the system checks the result cache in the memory firstly. If the results of q are found in the result cache, the system returns the cached results of q to the user directly. If the results of q are not found in the result cache, the system retrieves the corresponding posting list of each term of the query in the memory cache. If the term is not found in the memory cache, the system retrieves the global inverted list on the HDD (or the SSD), and then sorts and obtains the documents and generates the snippet. Finally, the system returns the results to the user. The memory cache is used to store all of the intermediate results, including result cache, posting list cache, snippet cache, and document cache. The storage medium is either HDD or SSD. The global inverted indexes are stored on the HDD or the SSD. When there is insufficient memory, the solution is to build memory swap with a secondary storage medium (HDD/SSD).

3.2. Two-level architecture

Because of excellent random read performance of the SSD, a two-level architecture uses the SSD as a secondary cache for the memory (Fig. 2). Due to the memory constraints, it is not possible to put all of the posting lists into the memory. Considering that more time is spent in reading the posting list from the HDD, an SSD layer is added between the memory and the HDD, which

Please cite this article in press as: X. Dong, et al., EDS: An Efficient Data Selection policy for search engine storage architectures, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.02.014

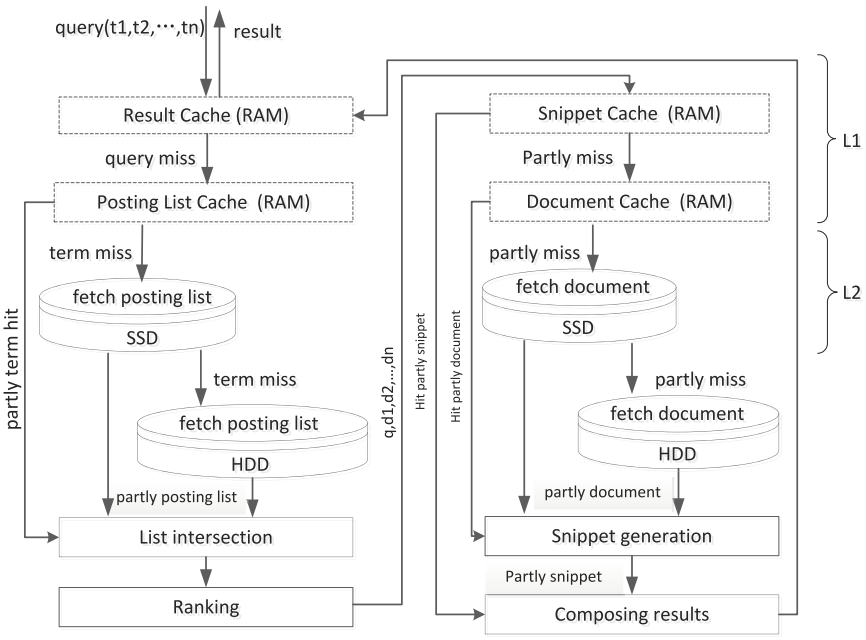


Fig. 2. Two-level search engine architecture.

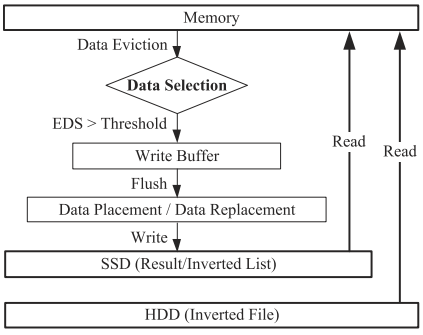


Fig. 3. The data management process of two-level search engine architecture.

is used to store warm posting lists. If the term is not found in the memory cache, the system retrieves the partly inverted list on the SSD. If the term is not found on the SSD, the system retrieves the global inverted list on the HDD. In the same way, some documents are stored on the SSD, reducing the time to obtain the document. Considering the shared data from the memory and the SSD, there are three caching schemes in the two-level cache, inclusive scheme, exclusive scheme, and hybrid scheme. With a page as the smallest cache unit, the three schemes can be described as follows.

Inclusive scheme. Whenever a page is in the memory, it is also cached on the SSD. That is to say, if the system caches a page in the memory, it should also write the page to the SSD.

Exclusive scheme. No page is stored on both the memory and the SSD at the same time. A page brought from the SSD to the memory is removed from the SSD, and vice versa.

Hybrid scheme. A page in the memory may or may not be cached on the SSD, depending on a criteria either set by the user or decided based on the current workload.

In this paper, we adopt the hybrid scheme, although it will bring some loss in the performance. The main reasons for adopting the hybrid scheme are as follows. If we use the inclusive scheme, the SSD and the memory share most of the cached data, which cannot bring the expected advantages of the SSD into full play. Also, if we take the exclusive scheme, the data should be removed when they are read from the SSD, which will cause a number of block erasure operations inside the SSD and shorten the life-span of the SSD. While in the hybrid scheme, all the hot data will be cached in the memory first. Once the cache in the memory is full, some of the least recently used data will be evicted and then written into the SSD according to some eviction policies. When the available capacity of the SSD is exhausted, the fresh data evicted from the memory will overwrite the cold data in the SSD. It is noted that if the data cached in the SSD is hit, they will be read from the SSD to the memory without deleting.

3.3. The data management process of the two-level architecture

Fig. 3 shows the data management process of the two-level search engine architecture which is described as the following steps.

First, when the user submits a query, the Lucene will read the data from the HDD (i.e., the inverted lists), and put it in the memory. Second, once the memory buffer overflows, the Lucene will eliminate some of the data (according to the cache replacement strategy). The data is eliminated by the filter of a data selection module. The threshold is the mean of the selected data, by screening ($EDS > Threshold$), the data will be added to the write buffer (called Write Buffer). Third, when the Write Buffer overflows, the Lucene will brush the data of the Writer Buffer into the SSD according to either the data placement strategy or the data replacement strategy. Fourth, as the data in the SSD is hit, the Lucene will read the corresponding data from the SSD to the memory.

4. Knapsack problem analysis for storage architectures

In this section, we describe the knapsack problem in the following different storage architectures: the HDD architecture, the SSD architecture, and the SSD-based hybrid storage architecture. First, we analyze the different situations of data access in these three architectures, and then we introduce the knapsack problem into the retrieval process. Finally, we give the derivation of the general formula.

Please cite this article in press as: X. Dong, et al., EDS: An Efficient Data Selection policy for search engine storage architectures, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.02.014

Table 1
Retrieval under different situations in HDD architecture.

Situation	Memory	HDD	Probability	Time cost
S_1	R		P_1	$T_1 = C_{mpr}$
S_2	I		P_2	$T_2 = C_{mpl} + C_0$
S_3		I	P_3	$T_3 = C_{hpl} + C_0$

Table 2
Retrieval under different situations in SSD architecture.

Situation	Memory	SSD	Probability	Time cost
S_1	R		P_1	$T_1 = C_{mpr}$
S_2	I		P_2	$T_2 = C_{mpl} + C_0$
S_3		I	P_3	$T_3 = C_{spl} + C_0$

4.1. Knapsack problem in HDD architecture

During the retrieval process based on the HDD-based storage architecture, there are three kinds of basic situations of data access in Table 1.

From above Table 1, we note that Memory and HDD denote where the data read, R denotes the results, and I denotes the inverted lists. Probability represents the probability that the corresponding situation takes place. Time Cost is the average retrieval time accordingly. In the calculation formula of the time cost, C_{mpr} is the time cost of obtaining the search results directly from Memory. C_{mpl} is the time cost of obtaining the inverted lists from Memory, and C_{hpl} is the time cost of obtaining the inverted lists from HDD. So, $C_0 = C_{rank} + C_{doc} + C_{snip}$. This formula describes the sum of the time cost of obtaining the search results by sorting, obtaining documents, and generating snippet. It is easy to know that $T_3 > T_k$ $(1 \le k \le 2)$. From the table, we get the average retrieval time as Formula (1).

$$AVG(T) = \sum_{i=1}^{3} T_i \times P_i. \tag{1}$$

In order to minimize the average retrieval time, we need to make full use of the advantages of the first-level cache by analyzing the different situations in Table 1. This means that the probabilities of "S_1" and "S_2" should be increased. Therefore, we analyze the time cost and the frequency of the single item, including the result and the inverted list. Suppose that there are n results and m inverted lists, each item will be accessed with a certain frequency. Considering that an item may either be cached in the Memory("MM" denotes memory.), or be obtained from the HDD indirectly, Formula (2) can then represent the mathematical expectations of the retrieval.

$$\begin{aligned} E(T) &= \sum_{i=1}^{n}(T_1 \cdot x_{iMM} + T_3 \cdot (1 - x_{iMM})) \cdot f_i \\ &\quad + \sum_{j=1}^{m}(T_2 \cdot x_{jMM} + T_3 \cdot (1 - x_{jMM})) \cdot f_j \\ &= \sum_{i=1}^{n} T_3 \cdot f_i - \sum_{i=1}^{n}[(T_3 - T_1) \cdot f_i \cdot x_{iMM}] \\ &\quad + \sum_{j=1}^{m} T_3 \cdot f_j - \sum_{j=1}^{m}[(T_3 - T_2) \cdot f_j \cdot x_{jMM}] \end{aligned} \tag{2}$$

$x_{i(storage)} = 1$ represents that item i is cached in the storage medium, otherwise $x_{i(storage)} = 0$. For example, $x_{iMM} = 1$ represents that item i is cached in main memory. For the purpose of minimizing the average retrieval time, we hope to keep the mathematical expectations of the access time as small as possible, and thus, we can get Formula (3).

$$\begin{aligned} &min\ AVG(T) \Leftrightarrow min E(T) \\ &\Leftrightarrow max \sum_{i=1}^{n}[(T_3 - T_1) \cdot f_i \cdot x_{iMM}] \\ &\quad + \sum_{j=1}^{m}[(T_3 - T_2) \cdot f_j \cdot x_{jMM}]. \end{aligned} \tag{3}$$

In the objective function, we can take $(T_3 - T_k)f_i$ as the value of the item. It means the time savings by the cache. x_i indicates whether the item is to be cached in the limited cache space or not. Thus, it is a 0–1 knapsack problem. Considering the items loaded into the knapsack, including n search results and m inverted lists, we can note that each item has its own size (W_i), access frequency (f_i), and retrieval time (as it can be seen in Table 1). The items also need to meet the limitations of the memory capacity C_{MM}. Due to the goals and these condition constraints above, the 0–1 knapsack problem can be described as a mathematical model (presented in the following Formula (4)).

$$\begin{aligned} &max \sum_{i=1}^{n}[(T_3 - T_1) \cdot f_i \cdot x_{iMM}] \\ &+ \sum_{j=1}^{m}[(T_3 - T_2) \cdot f_j \cdot x_{jMM}] \\ &s.t. \sum_{i=1}^{n} W_i \cdot x_{iMM} + \sum_{j=1}^{m} W_j \cdot x_{jMM} \le C_{MM} \\ &x_{iMM} = 0\ or\ 1, \quad (1 \le i \le n) \\ &x_{jMM} = 0\ or\ 1, \quad (1 \le j \le m). \end{aligned} \tag{4}$$

For the unbounded knapsack problem, the greedy algorithm can achieve the optimal solution. In this model, the size of a single item is far smaller than the capacity of the allocated memory. Thus, the greedy algorithm can closely approximate the optimal solution. Besides, the greedy algorithm is famous for its time efficiency. Therefore, for saving the computing cost, we use the greedy algorithm to solve the knapsack problem. First, we sort the items in ascending order according to the value per unit (UV), which is shown in the following Formula (5).

$$UV = \frac{(T_3 - T_k)f_i}{W_i} \quad (1 \le k \le 2). \tag{5}$$

4.2. Knapsack problem in SSD architecture

Based on the SSD storage architecture, three kinds of basic situations of data access are presented in Table 2.

From above Table 2, we can notice that the SSD denotes where the data read. In the calculation formula of the time cost, C_{spl} signifies the time cost of obtaining the correlative inverted lists from the SSD. Again, $C_0 = C_{rank} + C_{doc} + C_{snip}$. The document fetching and the snippet generation can be processed on the SSD. According to the character of the SSD, the time cost of fetching a document C_{doc} and generating a snippet C_{snip} can be decreased so as to reduce the total time cost.

In similar way, we complete the same steps in Section 4.1. In contrast, the object is converted from the HDD to the SSD. Also, we can use the model of 0–1 knapsack problem (presented in Formula (4)) and the value per unit (presented in Formula (5)).

4.3. Knapsack problem in hybrid architecture

In view of our proposed hybrid storage architecture, five kinds of basic situations of data access are displayed in Table 3.

To save the amount of the memory space in the hybrid architecture, only a part of the results are stored on the SSD. From

Please cite this article in press as: X. Dong, et al., EDS: An Efficient Data Selection policy for search engine storage architectures, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.02.014

Table 3
Retrieval under different situations in hybrid architecture.

Situation	Memory	SSD	HDD	Probability	Time cost
S_1	R			P_1	$T_1 = C_{mpr}$
S_2		R		P_2	$T_2 = C_{spr}$
S_3	I			P_3	$T_3 = C_{mpl} + C_0$
S_4		I		P_4	$T_4 = C_{spl} + C_0$
S_5			I	P_5	$T_5 = C_{hpl} + C_0$

Table 4
Cost computations in the cache of search engine.

Notation	Computation
C_{hpl}	$D_{seek} + D_{rotation} + D_{read} \times \mid I_j \mid \times S_p \div D_{block}$
C_{spl}	$S_{read} \times \mid I_j \mid \times S_p \div S_{block}$
C_{rank}	$CPU_{scoring} \times \sum_{ti \in q}(\mid I_j \mid \times S_p)$
C_{doc}	$D_{seek} + D_{rotation} + D_{read} \times \mid d_{top} \mid \div D_{block}$
C_{Sdoc}	$S_{read} \times \mid d_{top} \mid \div S_{block}$
C_{snip}	$CPU_{snippet} \times \mid d \mid$
C_{spr}	$S_{read} \times \mid R_{top} \mid \times S_r \div S_{block}$
C_{mpl}	$M_{read} \times \mid I_j \mid \times S_p \div D_{block}$
C_{mpr}	$M_{read} \times \mid R_{top} \mid \times S_r \div S_{block}$

Table 3, we see that C_{spr} is the time cost of obtaining the search results directly from the SSD. It is easy to know that $T_5 \geq T_k$ $(1 \leq k \leq 4)$. The average retrieval time is revealed in the following Formula (6).

$$AVG(T) = \sum_{i=1}^{5} T_i \times P_i. \tag{6}$$

The advantages of the hybrid cache made full use of the system for minimizing the average retrieval time via analyzing the different situations in Table 3. It means that the probability of "S_1", "S_2", "S_3", and "S_4" should be increased. Similarly, considering that an item may either be cached in the MM or in the SSD, or be obtained from the HDD indirectly, Formula (7) can then represent the mathematical expectations of the retrieval.

$$\begin{aligned} &E(T) \\ &= \sum_{i=1}^{n}(T_1 x_{iMM} + T_2 x_{iSSD} + T_5(1 - x_{iMM} - x_{iSSD}))f_i \\ &\quad + \sum_{j=1}^{m}(T_3 x_{jMM} + T_4 x_{iSSD} \\ &\quad + T_5(1 - x_{jMM} - x_{jSSD}))f_j \\ &= \sum_{i=1}^{n} T_5 f_i - \sum_{i=1}^{n}[(T_5 - T_1)f_i x_{iMM} + (T_5 - T_2)f_i x_{iSSD}] \\ &\quad + \sum_{j=1}^{m} T_5 f_j - \sum_{j=1}^{m}[(T_5 - T_3)f_j x_{jMM} \\ &\quad + (T_5 - T_4)f_j x_{jSSD}]. \end{aligned} \tag{7}$$

In order to minimize the average retrieval time, the mathematical expectations of the access time should be small as much as possible. And then the Formula (8) can be got.

$$\begin{aligned} &min\, AVG(T) \Leftrightarrow min E(T) \\ &\Leftrightarrow max \sum_{i=1}^{n}[(T_5 - T_1) \cdot f_i \cdot x_{iMM} + (T_5 - T_2) \cdot f_i \cdot x_{iSSD}] \\ &\quad + \sum_{j=1}^{m}[(T_5 - T_3) \cdot f_j \cdot x_{jMM} + (T_5 - T_4) \cdot f_j \cdot x_{jSSD}]. \end{aligned} \tag{8}$$

In the objective function, we can take $(T_5 - T_k)f_i$ as the value of the item. Similarly, it can be converted to a multi-knapsack problem. Considering different cache spaces of the Memory and the SSD, we build a mathematical model of a multi-knapsack problem. The items loaded into the knapsacks include n search results and m inverted lists. Each item has its own size (W_i), access frequency (f_i), and retrieval time (as it can be seen in Table 3). Also, the items need to meet the limitations of the memory capacity C_{MM} and the SSD capacity C_{SSD}. There is no intersection between the set of items cached in the memory and the SSD. According to above goals and these condition constraints, the multi-knapsack problem can be described as a mathematical model in Formula (9).

$$\begin{aligned} &max \sum_{i=1}^{n}[(T_5 - T_1)f_i x_{iMM} + (T_5 - T_2)f_i x_{iSSD}] \\ &+ \sum_{j=1}^{m}[(T_5 - T_3)f_j x_{jMM} + (T_5 - T_4)f_j x_{jSSD}] \\ &s.t. \sum_{i=1}^{n} W_i x_{iMM} + \sum_{j=1}^{m} W_j x_{jMM} \leq C_{MM} \\ &\sum_{i=1}^{n} W_i x_{iSSD} + \sum_{j=1}^{m} W_j x_{jSSD} \leq C_{SSD} \\ &x_{iMM} + x_{iSSD} = 0 \text{ or } 1 \quad (1 \leq i \leq n) \\ &x_{jMM} + x_{jSSD} = 0 \text{ or } 1 \quad (1 \leq j \leq m) \\ &x_{iMM} = 0 \text{ or } 1, \qquad x_{iSSD} = 0 \text{ or } 1 \quad (1 \leq i \leq n) \\ &x_{jMM} = 0 \text{ or } 1, \qquad x_{jSSD} = 0 \text{ or } 1 \quad (1 \leq j \leq m). \end{aligned} \tag{9}$$

In this model, the size of a single item is far less than the cache capacity of the Memory and the SSD. Similarly, we sort the items in ascending order according to the value per unit in Formula (10).

$$UV = \frac{(T_5 - T_k)f_i}{W_i} \quad (1 \leq k \leq 4). \tag{10}$$

5. Efficient data selection in storage architectures

In this section, we first define a variety of time costs in the query process, and then we give detailed derivation steps of the EDS in different storage architectures and describe the EDS algorithm. Finally, we compare and analyze the results of the derivation.

5.1. Definitions of saving time terms

In order to explore the changed rule, the time cost associated with each query step is computed by employing the formulas shown in Table 4. As it can be seen from this table, D_{seek} and $D_{rotation}$ represent the seek time and the rotational latency of the HDD respectively. D_{read}, S_{read} and M_{read} mean the time cost of obtaining one block of data from the HDD, the SSD and the Memory, respectively. D_{block} is the block size of the HDD, and S_{block} is the block size of the SSD. I_i represents the number of *DocId* of the ith posting list and S_p is the size of the storage per *DocId*. $CPU_{scoring}$ and $CPU_{snippet}$ represent the scoring cost and the snippet generation cost respectively. d_{top} represents the size of the highest scoring document, and d is the number of documents. R_{top} represents the highest scoring result. And finally, S_r is the average size per result.

5.2. Derivation of EDS in storage architectures

Compared with the inverted list entries, the result entries are quite small and similar in size, so we can take common policy to deal with the results. At the same time, we need some special selection policies for the inverted list. Therefore, we divide the effective values into two categories: the effective values of the result entries and the effective values of the inverted list entries.

Please cite this article in press as: X. Dong, et al., EDS: An Efficient Data Selection policy for search engine storage architectures, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.02.014

Table 5
The time saving with two types of knapsack in HDD.

Notation	Type	Time saving
Saving1	$T_3 - T_1$	$C_{hpl} + C_0 - C_{mpr} \approx C_{hpl} + C_0$
Saving2	$T_3 - T_2$	$C_{hpl} - C_{mpl} \approx C_{hpl}$

Based on Formula (5) and Formula (10), we find that the following three factors are associated with the value of UV: the saving time ($T_5 - T_k$), the access frequency (f_i) and the item size (W_i). We set EDS as the effective values per unit, and put the larger ones into the knapsack presented in the following Formula (11):

$$EDS = \frac{Saving \times f_i}{W_i}. \tag{11}$$

Based on different storage architectures, the EDS can be analyzed from three aspects: First, according to the time-consuming listed in Table 1, we suppose that all the result entries and part of the inverted lists entries are stored in the memory. By considering the differences of the time saving, we separate these two types of items, and get two knapsack problems (i.e., the results knapsack and the inverted lists knapsack). The time saving for each query step is deduced using the formulas shown in Table 5.

By calculating the parameters of the hardware and analyzing the query log, we find that the time consumption of C_{mpr} and C_{mpl} can be negligible relative to the time of reading the same result and inverted list from the HDD. Therefore, the value of Saving1 is approximately equal to $C_{hpl} + C_0$, and the value of Saving2 is approximately equal to C_{hpl}. By substituting C_{hpl} or $(C_{hpl} + C_0)$ into Formula (11), we can obtain the Formulas (12) and (13), respectively; where, EDS_{PL} is the EDS of the inverted lists and EDS_R is the EDS of the results in the HDD.

$$\begin{aligned} EDS_{PL} &= \frac{Freq \times C_{hpl}}{W_j} \\ &= \frac{Freq(D_{seek} + D_{rotation} + D_{read} \times |I_j| \times S_p / D_{block})}{W_j} \\ &= (D_{seek} + D_{rotation})\frac{Freq}{W_j} + \frac{Freq \times D_{read} \times |I_j| \times S_p}{W_j \times D_{block}} \\ &= (D_{seek} + D_{rotation})\frac{Freq}{W_j} + Freq \times \frac{D_{read}}{D_{block}} \\ &= C_1 \frac{Freq}{W_j} + C_2 Freq \\ &\left(C_1 = D_{seek} + D_{rotation}, C_2 = \frac{D_{read}}{D_{block}}\right) \end{aligned} \tag{12}$$

$$\begin{aligned} EDS_R &= \frac{Freq}{W_i} \times (C_{hpl} + C_0) \approx \frac{Freq}{W_i} \times (C_{hpl} + C_{doc}) \\ &= \frac{Freq \times (D_{seek} + D_{rotation} + D_{read} \times |I_j| \times S_p \div D_{block})}{W_i} \\ &\quad + \frac{Freq \times (D_{seek} + D_{rotation} + D_{read} \times |d_{top}| \div D_{block})}{W_i} \\ &= 2(D_{seek} + D_{rotation})\frac{Freq}{W_i} + \frac{Freq \times D_{read} \times |I_j| \times S_p}{W_i \times D_{block}} \\ &\quad + \frac{Freq \times D_{read} \times d \cdot K \div D_{block}}{W_i} \\ &= (2C_1 + C_2 W_j + C_3) \times \frac{Freq}{W_i} \quad (C_3 = d \times K \times C_2) \\ &= (C_6 + C_2 W_j) \times \frac{Freq}{W_i} \quad (C_6 = 2C_1 + C_3). \end{aligned} \tag{13}$$

Table 6
The time saving with two types of knapsack in SSD.

Notation	Type	Time saving
Saving1	$T_3 - T_1$	$C_{spl} + C_0 - C_{mpr} \approx C_{spl} + C_0$
Saving2	$T_3 - T_2$	$C_{spl} - C_{mpl} \approx C_{spl}$

Table 7
The time saving with each type of knapsack.

Notation	Type	Time saving
Saving1	$T_5 - T_1$	$C_{hpl} + C_0 - C_{mpr} \approx C_{hpl} + C_0$
Saving2	$T_5 - T_2$	$C_{hpl} + C_0 - C_{spr} \approx C_{hpl} + C_0$
Saving3	$T_5 - T_3$	$C_{hpl} - C_{mpl} \approx C_{hpl}$
Saving4	$T_5 - T_4$	$C_{hpl} - C_{spl} \approx C_{hpl}$

Second, according to the time-consuming listed in Table 2, and the same above assumption, the time saving for each query step is deduced using the formulas shown in Table 6.

As we have discussed before, the time consumption of C_{mpr} and C_{mpl} can be negligible relative to the time of reading the same result and inverted list from the HDD. Therefore, the value of Saving1 is approximately equal to $C_{spl} + C_0$, and the value of Saving2 is approximately equal to C_{spl}. By substituting C_{spl} or $(C_{spl} + C_0)$ into Formula (11), we can obtain the Formulas (14) and (15), respectively; where, EDS_{SPL} represents the EDS of the inverted lists and EDS_{SR} represents the EDS of the results in the SSD.

$$\begin{aligned} EDS_{SPL} &= \frac{Freq \times C_{spl}}{W_j} \\ &= \frac{Freq \times (S_{read} \times |I_j| \times S_p \div S_{block})}{W_j} \\ &= \frac{S_{read}}{S_{block}} \times Freq = C_4 Freq \quad \left(C_4 = \frac{S_{read}}{S_{block}}\right) \end{aligned} \tag{14}$$

$$\begin{aligned} EDS_{SR} &= \frac{Freq}{W_i}(C_{spl} + C_0) \approx \frac{Freq}{W_i}(C_{spl} + C_{Sdoc}) \\ &= \frac{Freq(S_{read}|I_j|S_p/S_{block})}{W_i} + \frac{Freq(C_{Sdoc})}{W_i} \\ &= (C_4 W_j + C_4 \times d \times K)\frac{Freq}{W_i} \\ &= (C_5 + C_4 W_j)\frac{Freq}{W_i} \quad (C_5 = d \times K \times C_4). \end{aligned} \tag{15}$$

Third, according to the time-consuming listed in Table 3, we suppose that part of the result entries and inverted lists entries are stored either in the memory or the SSD. In the hybrid storage architecture, we find that there are big differences between the items cached in the memory and those cached in the SSD. Therefore, reading data from the memory is significantly faster than that from the SSD. We can approximate that all of the items cached in the memory have higher UV values than the items cached in the SSD. Therefore, we can divide the multi-knapsack problem into two stages, the memory knapsack problem and the SSD knapsack problem. Similarly, by considering the differences in the time saving and the size between the results and the inverted lists, we separate these two types of items, and get two knapsack problems. In this way, the multi-knapsack problem is transformed into four basic 0–1 knapsack problems. The time saving is deduced using the formulas shown in the following Table 7.

In the same way, the value of Saving1 is approximately equal to $C_{hpl} + C_0$, and the value of Saving3 is approximately equal to C_{hpl}. The time of reading the results and the inverted lists from the SSD is longer than that from the memory, but due to the high speed reading performance of the SSD, there are almost two orders of magnitude better than the time of reading the inverted list from the

Please cite this article in press as: X. Dong, et al., EDS: An Efficient Data Selection policy for search engine storage architectures, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.02.014

HDD. Consequently, the value of Saving2 is approximately equal to $C_{hpl}+C_0$, and the value of Saving4 is approximately equal to C_{hpl}. We also find that the saving time of reading the inverted lists is C_{hpl} (i.e., approximately through observation, it is the same as the values in Saving3 and Saving4). Similarly, the saving time of reading the result is $C_{hpl} + C_0$. So, C_{hpl} is the key factor of the saving time of reading the inverted lists, and $C_{hpl}+C_0$ is the key factor of the saving time of reading the result. By substituting C_{hpl} or $(C_{hpl} + C_0)$ into Formula (11), we can also obtain the previous two Formulas (12) and (13), respectively.

As viewed from Formula 12, we can compute the value of C_1 and C_2 according to the parameters of the hard disk. (For example, when $D_{seek} = 8.5$ ms, $D_{rotation} = 4.2$ ms, $D_{read} = 4.88$ ms, $S_p =$ 8 bytes, and $D_{block} = 512$ bytes, then $C_1 = 12.7$ and $C_2 = 0.0095$). In Formula 13, we find that the time consumption of C_{rank} and C_{snip} can be negligible relative to the time of C_{doc}. Therefore, the value of C_0 is approximately equal to C_{doc}; where, K represents the average size per document. When the size of the result is fixed (as W is a constant), then, EDS_R is a function of $Freq$. (For example, when $d = 10, K = 8$ kb, and $W = 4$ kb, then $C_3 = 780.8$ and $EDS_R = 0.22Freq$.)

5.3. Efficient data selection algorithm

The efficient data selection algorithm is designed to ensure that the cached data has a high hit ratio and low query latency under different storage architectures. Algorithm 1 describes three strategies under different storage architectures. In the SSD-based hybrid architecture, the system sorts the term in query log by the EDS_{PL} in descending order and fills up the memory with the posting lists of the sorted term. We assign EDS_{PL} of the last term filled in the memory as a minimum value $EDS_{\min}$. If the EDS_{PL} of the next term is greater than the $EDS_{\min}$ in the query, we use posting list of the term instead of $EDS_{\min}$. And then the system sorts and updates the $EDS_{\min}$. At the same time, the relationship between the $EDS_{\min}$ and the threshold is also considered. If the evicted $EDS_{\min}$ is greater than the threshold, the posting list of the $EDS_{\min}$ is brushed into the SSD. In the SSD architecture, the key difference is that the term is sorted and the posting list of the term is replaced by frequency. In the HDD architecture, here similar to the case of hybrid architecture, the sorting and the replacement are based on the EDS_{PL} value.

5.4. Comparison of EDS

In this subsection, our goal is to discover some factors by comparing the above EDS values. And then, by analyzing we can discover which one is the dominant among these factors. The contrast includes two aspects: the inverted lists and the results.

First, by considering the EDS values of the inverted lists (Formula (12) and Formula (14)), we can note that EDS_{PL} is only related to the frequency (Formula (14)). As a consequence, the frequency is a key factor in the SSD storage architecture. In the HDD storage architecture (Formula (12)), there are two relation factors: $Freq/W_j$ and $Freq$. We set $C_1/W_j = C_2$ (for example, $C_1 = 12.7$ ms, and $C_2 = 4.88$ ms/512, so, $W_j = 1332$). Obviously, in this example, when $W_j \leq 1332$, $Freq/W_j$ plays a vital role, but when $W_j > 1332$, $Freq$ plays an important role. Therefore, the average length of the inverted lists is an influence factor for search engine cache.

Second, by considering the EDS values of the results (Formula (13) and Formula (15)), we can note that EDS_R and EDS_{SR} are also closely related to the frequency. When the size of the results is fixed, then W_i is a constant. But there are some slightly differences between EDS_R and EDS_{SR}, because the values of C_6/C_5 or C_2/C_4 are all approximated to 600 in our experiments. Therefore, $EDS_RR = 600EDS_{SR}$. That is to say, relative to the HDD, the performance of the SSD is closer to the performance of the memory. Similarly, when we set $C_6 = C_2W_j$, we can attain the value of W_j, which is approximated to 80,000 in our experimental environment. As a result, when $W_j \geq 80{,}000$, the fetch time of the inverted lists plays a vital role. But when $W_j < 80{,}000$, the fetch time of documents plays a partly important role.

The above examples can be simplified as shown in Fig. 4. Fig. 4 expresses the impact of the average length of the inverted lists for the inverted lists cache and the results cache. Therefore, we can obtain the following conclusion: in the SSD architecture, the higher $Freq$ of the items, the better the benefit of the inverted list; and the longer the inverted list, the better benefit of the results cache.

Algorithm 1 Efficient data selection

Input: $D_{seek}, D_{rotation}, D_{read}, D_{block}$; Query Log; & Document Data Set;
Output: Selected efficient data in memory (E_M)& on SSD (E_S)

Count the number of occurrences of each query term in Query Log: Freq
Count the number of documents, where each query term is included in Document Data Set: W_j
Calculate C_1 and C_2. $C_1 = D_{seek} + D_{rotation}$, and $C_2 = D_{read}/D_{block}$
Calculate EDS_{PL} of each term, $EDS_{PL} = C_1Freq/Wj + C_2Freq$.
Calculate Threshold (mean of EDS_{PL}), $Threshold = \sum_{j=1}^{m} EDS_{PL}/m$
Create an inverted index for Document Data Set and use Query Logs to query.
Sort the term by EDS_{PL} in descending order: $Term_{EDS}$.
Sort the term by Freq in descending order: $Term_{Freq}$.
switch (architecture)
case "2L: SSD-based hybrid":
E_M = fill up the memory with the posting lists of $Term_{EDS}$. Assign EDS_{PL} of the last term filled in the memory as a minimum value EDS_{min}.
  while EDS_{PL} of next term > EDS_{min} **do**
Use posting list of the term instead of EDS_{min}
    If evicted EDS_{min} > $Threshold$ **then**
E_S = brush the posting list of the EDS_{min} into the SSD
    end if
Update EDS_{min}: minimum values of $Term_{EDS}$ queue in memory (E_M).
  end while
break;
case "1L: Memory+SSD":
E_M = fill up the memory with the posting lists of $Term_{Freq}$. Assign Freq of the last term filled in the memory as a minimum value $Freq_{min}$.
  while Freq of next term > $Freq_{min}$ **do**
Use posting list of the term instead of $Freq_{min}$
Update $Freq_{min}$: minimum values of $Term_{Freq}$ queue in memory (E_M).
  end while
break;
case "1L: Memory+HDD":
E_M = fill up the memory with the posting lists of $Term_{EDS}$. Assign EDS_{PL} of the last term filled in the memory as a minimum value EDS_{min}.
  while EDS_{PL} of next term > EDS_{min} **do**
Use posting list of the term instead of EDS_{min}
Update EDS_{min}: minimum value of $Term_{EDS}$ queue in memory (E_M).
  end while
return E_M,E_S

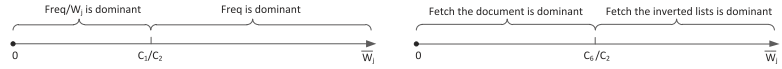
6. Performance evaluation

In this section, our evaluation includes three aspects. The first one is comparing the hit ratios of several proposed algorithms. The second is verifying that our proposed policy can improve the retrieval performance of the search engines, including the HDD architecture, the SSD architecture, and the SSD-based hybrid storage architecture. The third is verifying that our proposed policy can also reduce the cost of the search engine servers. We preserve the notations of the previous sections here.

6.1. Experimental settings

Table 8 summarizes the experimental platform specifications. In our experiments, the total data size from enwiki data set is 8.9 GB, the number of the total documents is 5,000,000, where the index size of which is 5.12 GB. The used query log was from AOL. We have prepared a set of sample queries from the AOL query log, which performed the index retrieval. Our simulative search engine is based on Lucene 3.5.0.

Please cite this article in press as: X. Dong, et al., EDS: An Efficient Data Selection policy for search engine storage architectures, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.02.014



(a) The average W_j for the inverted lists cache. (b) The average W_j for the results cache.

Fig. 4. The impact of the average length of the inverted lists.

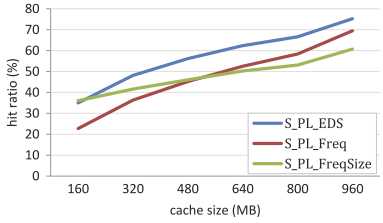


Fig. 5. The hit ratio comparison in HDD (static).

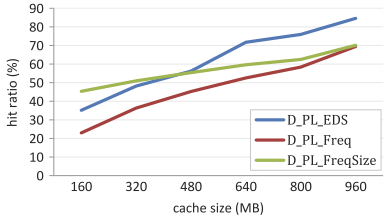


Fig. 6. The hit ratio comparison in HDD (dynamic).

Table 8
The experiment platform specifications.

Experiment platform environment	
IR Tool	Lucene 3.5.0
Data Set	enwiki-20090805-pages-articles.xml
Query Log	AOL-user-ct-collection
SSD	Samsung SSD 840 Series 120 GB
HDD	Seagate ST2000DM001-1CH164 2 TB
OS	Windows 7/Ubuntu 12.04
CPU/RAM	Inter(R) Xeon (R) CPU E3 1230 V2/16G

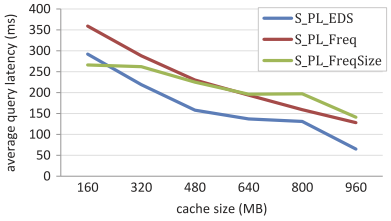


Fig. 7. Performance comparison in HDD (static).

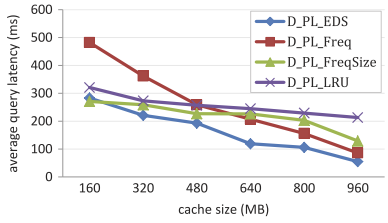


Fig. 8. Performance comparison in HDD (dynamic).

6.2. Hit ratio

There are two existing static query inverted list caching policies, which we refer them as S_PL_Freq and S_PL_FreqSize, respectively. The static implementation process of the S_PL_Freq (S_PL_FreqSize) is as follows. We sort the term of the query log by frequency (frequency/size) in descending order, and fill up the memory with the posting lists of the sorted term all at once. These posting lists will not be updated. In addition, we refer to the new static caching policy proposed in Section 5.2, as S_PL_EDS.

In our experiments, the number of total documents is 1,000,000, the query count is 100,000, and the cache size ranges from about 160 to 960 MB. Fig. 5 shows the cache hit ratio comparison between S_PL_Freq, S_PL_FreqSize, and S_PL_EDS. It can be seen from Fig. 5 that the hit ratio will be increased with the increasing of the cache capacity at a certain range. Also, it can be seen that S_PL_FreqSize, which tends to cache the popular terms with short posting lists, has a higher cache hit ratio than S_PL_Freq in the previous stage. However, if we increase the cache capacity continuously, the hit ratio of S_PL_Freq exceeds that of S_PL_FreqSize at a later stage. The reason is that S_PL_Freqsize loads larger posting lists after 480 MB, and loses the advantage of having more posting lists. In the static case, S_PL_EDS has the highest cache hit ratio among the three policies. The reason is that S_PL_EDS combined the advantage of both S_PL_Freq and S_PL_FreqSize.

The dynamic cache version of S_PL_Freq, S_PL_FreqSize, and S_PL_EDS try to capture the recently data accessed. We refer them as D_PL_Freq, D_PL_FreqSize and D_PL_EDS, respectively. The dynamic implementation process of the D_PL_Freq (D_PL_FreqSize, D_PL_EDS) is a real-time update of the posting lists of the memory cache by frequency (frequency/size, EDS_{PL}).

Fig. 6 shows the hit ratio of D_PL_Freq, D_PL_FreqSize, and D_PL_EDS in the dynamic case. As seen from this figure, D_PL_FreqSize always has a higher hit ratio than D_PL_Freq. The hit ratio of D_PL_FreqSize is too higher than that of D_PL_EDS in the previous stage. However, at a later stage, the hit ratio of D_PL_EDS exceeds the hit ratio of D_PL_FreqSize. The reason is that the advantages of the caching popular terms have been gradually disappearing. We can compare the changes in the hit ratios of the three policies between the dynamic and the static cases, respectively. Our proposed D_PL_EDS policy improves the hit ratio by 20.04% in average compared with the D_PL_Freq and D_PL_FreqSize policies.

6.3. Retrieval performance on HDD

Our experimental setup is the same as for the last tests (Section 6.2). Fig. 7 shows the average query time on the HDD. The query latency of the three selected caching policies is consistent with the tradition holds: when one policy has a higher cache hit ratio than the others, its query latency is also shorter than the others. On the whole, the S_PL_EDS policy has the best query latency compared with the S_PL_FreqSize and S_PL_Freq policies. Specifically, we can note that the average query time of S_PL_EDS is slightly worse than the average query time of S_PL_FreqSize at 160 MB cache memory because the latter has much higher cache hit ratio.

Fig. 8 shows the average query time on the HDD in the dynamic case. Compared with the static case, the average query latency of the dynamic case has the same trends. In comparison with LRU, the average response latency is reduced by 31.98% in the EDS. Since the search content is dispersed and the periodicity is not strong, the LRU has no advantage.

6.4. Retrieval performance on SSD

In our experiments, the number of the total documents is 1,000,000, the query count is 100,000, and the cache size ranges

Please cite this article in press as: X. Dong, et al., EDS: An Efficient Data Selection policy for search engine storage architectures, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.02.014

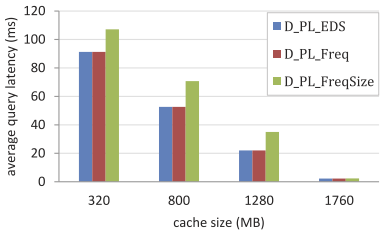


Fig. 9. Performance comparison in SSD (dynamic).

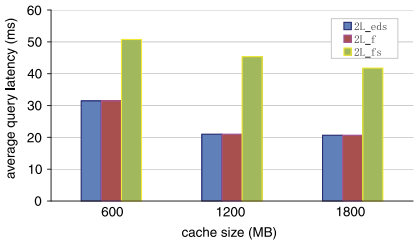


Fig. 10. Average query time with filter.

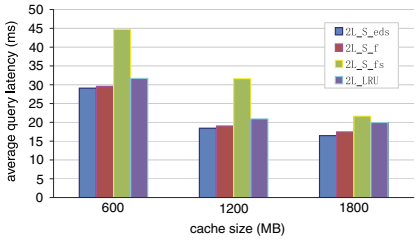


Fig. 11. Average query time with static cache.

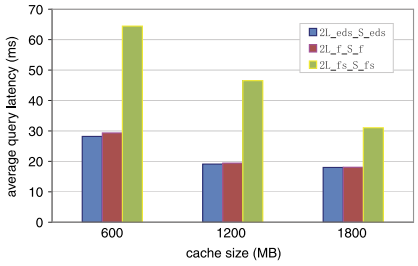


Fig. 12. Average query time with static cache and filter.

from about 320 to 1760 MB. Fig. 9 shows the average query time on the SSD in the dynamic case. The average query time has followed the same trend in the static case. According to Formula (14), EDS_{SPL} only related to the frequency, so D_PL_EDS is the same as D_PL_Freq. However, D_PL_FreqSize becomes poor in terms of the query latency. The reason is that the benefit brought by the higher hit ratio of D_PL_FreqSize is watered down by the fewer sequential read savings caused by the short posting lists. Compared with FreqSize, the average response latency is reduced by 28.72% in EDS.

6.5. Retrieval evaluation on hybrid architecture

In our experiments, the number of the total documents is 1,000,000, the query count is 100,000, and the cache size ranges from about 600 to 1800 MB. In two-level architecture, the data selection method is relatively complicated. In the dynamic case, the data selection (filtering) strategy can be implemented when the data flow from the memory to the SSD. If the data is not filtered, it is usually the secondary LRU method: 2L_LRU that will be implemented. This implementation is very uneconomical because the posting lists takes up plenty of the SSD space, bringing a certain degree of wear and tear on the SSD. If the data is filtered, we can use three strategies (i.e., EDS, Freq and FreqSize). We refer them as 2L_eds, 2L_f and 2L_fs, respectively, as shown in Fig. 10. Considering that the selected data will be written to the SSD, our proposed EDS method only adopt the frequency factor in this experiment. Fig. 10 shows that the average query time of 2L_eds (i.e., 2L_f) is significantly better than that of 2L_fs. The reason is a result of that the cache terms have high frequency saving query times.

In order to improve the efficiency of the retrieval, we have also introduced the static cache memory and SSD. The benefit of the static cache will not be mentioned in this paper since it has been confirmed in our previous work [10]. Like other relevant literatures, our previous work also used a single index (Freq or Freq/size) to measure the effectiveness of the cache. We have extended our EDS way in this experiment. Our work is to select the data joined in the static cache, where both the latest data and the hottest data will be loaded in the static cache. Once those data are added, they cannot be changed. Three selection ways were used in this case: the EDS, the Freq and, the FreqSize; which we refer them as 2L_S_eds, 2L_S_f, and 2L_S_fs, respectively, as shown in Fig. 11. Fig. 11 shows that, in the case of the static cache, the average query time is reduced more than in the case of without the static cache. As viewed from this figure, the 2L_S_eds takes the least amount of the average query time in all cache policies.

When we filter the obsoleted data from the memory and select the data which will be added to the static cache, the corresponding ways have three kinds. We refer them as 2L_eds_S_eds, 2L_f_S_f, and 2L_fs_S_fs, respectively, as displayed in Fig. 12.

Fig. 12 shows that the average query time of 2L_eds_S_eds is slightly better than that of 2L_f_S_f. We believe that the reason is that the static cache has achieved some benefits for the 2L_eds_S_eds. In this experiment, the average response latency is reduced by 23.24% in the EDS compared to FreqSize.

6.6. Cost performance evaluation

In this experiment, the number of the total indexed documents is 5 million, and the query count is 100,000. In Fig. 13, "1L_HDD" denotes one-level cache using the memory and the HDD, "1L_SSD" denotes one-level cache using the memory and the SSD, and "2L_Hybrid" denotes two-level cache taking the memory and the SSD as the cache. "HDD" denotes that the index files are stored on the HDD, and "SSD" denotes that the index files are stored on the SSD. Fig. 13 indicates that our proposed SSD-based hybrid storage architecture demonstrates the best performance comparing to the one-level cache architecture with index files stored on the HDD or the SSD. In Fig. 13, 1L_SSD is better than 1L_HDD. The reason is that we put the partly inverted indexes and documents on the SSD. The rapid reading performance of the SSD improves the time of querying and obtaining documents.

In Fig. 14, "MM" denotes the memory. Fig. 14 represents the average response time under the same cost conditions (i.e., one used 0.4G memory while the other used 0.2G memory and 1.4G SSD). By considering that the cost per GB of the memory is 7 times than that of the SSD in the current market, so we can store seven times of the amount of the result cache and the inverted lists on the SSD in an equivalent memory. The experimental results show that the two-level cache architecture is superior to the one-level cache architecture obviously. Therefore, we can reduce the capacity of

Please cite this article in press as: X. Dong, et al., EDS: An Efficient Data Selection policy for search engine storage architectures, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.02.014

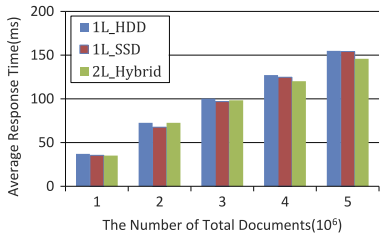


Fig. 13. 1L cache and 2L cache comparison.

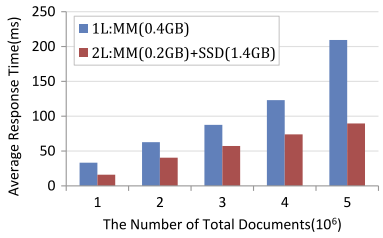


Fig. 14. Response time under equal cost conditions.

the memory and enlarge the capacity of the SSD without having any performance degradation in the two-level cache architecture.

The experimental results have shown that our proposed EDS policy is better than the Freq and the FreqSize policies in the performance for different architectures. We believe that there are two main reasons for this achievement. First, the EDS policy has the highest cache hit ratio. Second, the EDS policy can be viewed as a reasonable compromise between these two policies. The value of the EDS depends on the parameters of the specific storage medium. The efficient data are always placed in the cache through the guidance of the query log and the EDS policy. The EDS policy can be used in distributed environment. Ideally, each node can add an SSD to enhance the processing speed of a single node, and thus the processing speed of the whole distributed system can be improved. However, the policy still need to consider the network I/O factors.

7. Conclusion and future work

In this paper, we have described three types of storage architectures for search engines, and then defined and analyzed the knapsack problem in different storage architectures. We have found some critical factors via derivation and comparison. Meanwhile, we have proposed an EDS policy placing the efficient data in either the memory or the SSD. Finally, the experimental results have demonstrated our proposed EDS policy.

There are several interesting problems that need further discussion. First, our work only considers the data selection policies of the result cache and the posting list cache. In practice, the cache management of the search engine also includes intersection cache, document cache, etc. The data selection policies of these caches are required further study in the future. Second, due to the introduction of the SSD in the cache system, the data placement and replacement policies also need to be considered based on the characteristics of the SSD, which can further improve the performance of the search engine.

Acknowledgments

This work is supported by the National Natural Science Foundation of China under grants 61173170, 61300222, 61370230, 61433006 and U1401258, and the Innovation Fund of Huazhong University of Science and Technology under grants 2015TS069 and 2015TS071, Science and Technology Support Program of Hubei Province under grant 2014BCH270 and 2015AAA013, and Science and Technology Program of Guangdong Province under grant 2014B010111007. Meikang Qiu is supported by the NSF 1457506. We sincerely thank the anonymous reviewers for their very comprehensive and constructive comments.

References

[1] S. Chen, P.B. Gibbons, S. Nath, Rethinking database algorithms for phase change memory, in: The 5th Biennial Conference on Innovative Data Systems Research, CIDR'11, Online Proceedings, Asilomar, CA, USA, 2011, pp. 21–31.
[2] J. Wang, E. Lo, M.L. Yiu, J. Tong, G. Wang, X. Liu, The impact of solid state drive on search engine cache management, in: The 36th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR'13, ACM, Dublin, Ireland, 2013, pp. 693–702.
[3] Releasing flashcache, 2015. https://github.com/facebook/flashcache/blob/master/doc/flashcache-sa-guide.txt.
[4] Microsoft azure to use ocz SSDs, 2015. http://www.storagelook.com/microsoft-azure-ocz-ssds/.
[5] Google plans to use intel SSD storage in servers, 2015. http://www.networkcomputing.com/storage/google-plans-to-use-intel-ssd-storage-in-servers/d/d-id/1067741.
[6] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, Y. Wang, Sdf: Software-defined flash for web-scale Internet storage systems, in: The 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'14, ACM, Salt Lake City, UT, USA, 2014, pp. 471–484.
[7] V. Kasavajhala, Solid state drive vs. hard disk drive price and performance study, Technical Report, Dell PowerVault Technical Marketing, 2011.
[8] Google platform, 2015. https://en.wikipedia.org/wiki/Google_platform.
[9] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, T. Wobber, Extending SSD lifetimes with disk-based write caches, in: The 8th USENIX Conference on File and Storage Technologies, FAST'10, USENIX, San Jose, CA, USA, 2010, pp. 101–114.
[10] R.X. Li, C.Z. Li, W.J. Xiao, H. Jin, H. He, X. Gu, K.M. Wen, Z.Y. Xu, An efficient SSD-based hybrid storage architecture for large-scale search engines, in: The 41st International Conference on Parallel Processing, ICPP'12, IEEE, Pittsburgh, PA, USA, 2012, pp. 450–459.
[11] E.P. Markatos, On caching search engine query results, Comput. Commun. 24 (1) (2001) 137–143.
[12] R. Ozcan, I.S. Altingovde, U. Ulusoy, Static query result caching revisited, in: The 17th International Conference on World Wide Web, WWW'08, ACM, Beijing, China, 2008, pp. 1169–1170.
[13] Q. Gan, T. Suel, Improved techniques for result caching in web search engines, in: The 18th International Conference on World Wide Web, WWW'09, ACM, Madrid, Spain, 2009, pp. 431–440.
[14] E. Rosas, N. Hidalgo, M. Marin, V.G. Costa, Web search results caching service for structured p2p networks, Future Gener. Comput. Syst. 30 (2014) 254–264.
[15] R.A. Baeza-Yates, F. Saint-Jean, A three level search engine index based in query log distribution, in: The 10th International Symposium on String Processing and Information Retrieval, SPIRE'03, Springer, Manaus, Brazil, 2003, pp. 56–65.
[16] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, F. Silvestri, The impact of caching on search engines, in: The 30th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR'07, ACM, Amsterdam, Netherlands, 2007, pp. 183–190.
[17] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, F. Silvestri, Design trade-offs for search engine caching, ACM Trans. Web 2 (4) (2008) 1–28.
[18] R. Ozcan, I.S. Altingovde, B.B. Cambazoglu, F.P. Junqueira, U. Ulusoy, A five-level static cache architecture for web search engines, Inf. Process. Manage. 48 (5) (2011) 828–840.
[19] A. Turpin, Y. Tsegay, D. Hawking, H.E. Williams, Fast generation of result snippets in web search, in: The 30th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR'07, ACM, Amsterdam, Netherlands, 2007, pp. 127–134.
[20] D. Ceccarelli, C. Lucchese, S. Orlando, R. Perego, F. Silvestri, Caching query-biased snippets for efficient retrieval, in: The 14th International Conference on Extending Database Technology, EDBT'11, ACM, Uppsala, Sweden, 2011, pp. 93–104.
[21] T. Fagni, R. Perego, F. Silvestri, S. Orlando, Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data, ACM Trans. Inf. Syst. 24 (2006) 51–78.
[22] D. Yuan, Y. Yang, X. Liu, J. Chen, A data placement strategy in scientific cloud workflows, Future Gener. Comput. Syst. 26 (1) (2010) 1200–1241.
[23] B.B. Cambazoglu, F.P. Junqueira, V. Plachouras, S.A. Banachowski, B. Cui, S. Lim, B. Bridge, A refreshing perspective of search engine caching, in: The 19th International Conference on World Wide Web, WWW'10, ACM, Raleigh, North Carolina, USA, 2010, pp. 181–190.

Please cite this article in press as: X. Dong, et al., EDS: An Efficient Data Selection policy for search engine storage architectures, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.02.014

[24] P. Saraiva, E. de Moura, R. Fonseca, J.W. Meira, B. Ribeiro-Neto, N. Ziviani, Rank-preserving two-level caching for scalable search engines, in: The 24th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR'01, ACM, New Orleans, Louisiana, USA, 2001, pp. 51–58.
[25] X. Long, T. Suel, Three-level caching for efficient query processing in large web search engines, in: The 14th international conference on World Wide Web, WWW'05, ACM, Chiba, Japan, 2005, pp. 369–395.
[26] J. Matthews, S. Trika, D. Hensgen, R. Coulson, K. Grimsrud, Intel turbo memory: Nonvolatile disk caches in the storage hierarchy of mainstream computer systems, TOS 4 (2) (2008) 1–24.
[27] R. Panabaker, Hybrid hard disk and readydrive technology: Improving performance and power for windows vista mobile pcs, in: Proc. of the Microsoft Win-HEC 2006, Microsoft, Los Angeles, CA, 2006, pp. 1–32.
[28] J. Suk, J. No, Y. Kim, Design and implementation of hybridfs, in: The 3rd IEEE International Conference on Computer Science and Information Technology, ICCSIT'10, IEEE, Amsterdam, Netherlands, 2010, pp. 501–505.
[29] J. No, Hybrid file system using nand-flash SSD, in: The 2011 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, CyberC'11, IEEE, Beijing, China, 2011, pp. 380–385.
[30] B. Kai, W. Meng, N. Hongshan, H. Wei, L. Bo, The optimization of the hierarchical storage system based on the hybrid SSD technology, in: The 2nd International Conference on Intelligent System Design and Engineering Application, ISDEA'12, IEEE, Sanya, China, 2012, pp. 1323–1326.
[31] H. Ishibuchi, Y. Tanigaki, N. Akedo, Y. Nojima, How to strike a balance between local search and global search in multiobjective memetic algorithms for multiobjective 0/1 knapsack problems, in: The IEEE Congress on Evolutionary Computation, CEC'13, IEEE, Cancun, Mexico, 2013, pp. 1643–1650.
[32] A. Sbihi, Adaptive perturbed neighbourhood search for the expanding capacity multiple-choice knapsack problem, JORS 64 (1) (2013) 1461–1473.
[33] B. Huang, Z. Xia, Allocating inverted index into flash memory for search engines, in: The 20th International Conference Companion on World Wide Web, WWW'11, ACM, Hyderabad, India, 2011, pp. 61–62.
[34] E. Chan, Y. Wang, W. Li, S. Lu, Movement prediction based cooperative caching for location dependent information service in mobile ad hoc networks, J. Supercomput. 59 (1) (2012) 297–322.

Xinhua Dong received his M.S. degree from School of Computer Science and Technology at Huazhong University of Science and Technology in 2008. Now he is a Ph.D. candidate in the Intelligent and Distributed Computing Laboratory, School of Computer Science and Technology, Huazhong University of Science and Technology. His research interests include cloud computing, information retrieval, and big data management.

Ruixuan Li received the B.S., M.S., and Ph.D. degrees from School of Computer Science and Technology at Huazhong University of Science and Technology in 1997, 2000, and 2004 respectively. He is currently a professor of School of Computer Science and Technology at Huazhong University of Science and Technology, and is the director of the Intelligent and Distributed Computing Laboratory. His research interests include cloud computing, big data, and system security.

Heng He received his M.S. degree from School of Computer Science and Technology at Huazhong University of Science and Technology in 2007. Now he is a Ph.D. candidate in the Intelligent and Distributed Computing Laboratory, School of Computer Science and Technology, Huazhong University of Science and Technology. His research interests include cloud computing, distributed simulation, and network security.

Xiwu Gu received his Ph.D. degrees from School of Computer Science and Technology at Huazhong University of Science and Technology in 2007. He is currently a lecturer of School of Computer Science and Technology at Huazhong University of Science and Technology. His research interests include distributed system, web service, and middleware.

Mudar Sarem received his B.S. degree in Electronics Engineering from Tishreen University, Syria, in 1989, M.S. and Ph.D. degree in Computer Science from Huazhong University of Science and Technology, China in 1997 and 2002, respectively. He is currently an Associate Professor with the School of Software Engineering, Huazhong University of Science and Technology, Wuhan, China. His research interests include computer graphics, multimedia databases, and image processing.

Meikang Qiu received B.E. and M.E. degree in engineering from Shanghai Jiao Tong University, China, and M.S. and Ph.D. degree in Computer Science from University of Texas at Dallas. He is currently an associate professor of department of Computer Science at Pace University, ACM/IEEE Senior member. His research interests include cloud computing, cyber security and privacy, big data and data analytic, Telehealth system, Embedded systems and Mobile systems.

Keqin Li received B.S. degree in Computer Science from Tsinghua University, China, in 1985, and Ph.D. degree in Computer Science from the University of Houston in 1990. He is currently a SUNY Distinguished professor of State University of New York at New Paltz, IEEE Fellow, and is Intellectual Ventures Endowed Visiting Chair Professor at Tsinghua University. His research interests are mainly in the areas of design and analysis of algorithms, parallel and distributed computing, and computer networking. His current research interests include lifetime maximization in sensor networks, file sharing in peer to peer systems, and cloud computing.

Please cite this article in press as: X. Dong, et al., EDS: An Efficient Data Selection policy for search engine storage architectures, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.02.014