WUJNS
Wuhan University Journal of Natural Sciences

Vol.12 No.5 2007 830-834

Article ID: 1007-1202(2007)05-0830-05
DOI 10.1007/s11859-007-0042-8

KDS-CM: A Cache Mechanism Based on Top-*K* Data Source for Deep Web Query

□ **KOU Yue[1], SHEN Derong[1], YU Ge[1†], LI Dong[2], NIE Tiezheng[1]**

1. College of Information Science and Engineering, Northeastern University, Shenyang 110004, Liaoning, China;

2. Business Software Division, Neusoft Group Limited Company, Shenyang 110004, Liaoning, China

Abstract: Caching is an important technique to enhance the efficiency of query processing. Unfortunately, traditional caching mechanisms are not efficient for deep Web because of storage space and dynamic maintenance limitations. In this paper, we present on providing a cache mechanism based on Top-*K* data source (KDS-CM) instead of result records for deep Web query. By integrating techniques from IR and Top-*K*, a data reorganization strategy is presented to model KDS-CM. Also some measures about cache management and optimization are proposed to improve the performances of cache effectively. Experimental results show the benefits of KDS-CM in execution cost and dynamic maintenance when compared with various alternate strategies.

Key words: cache; Top-*K*; Deep Web; data reorganization; cache management and optimization

CLC number: TP 393

Received date: 2007-02-23

Foundation item: Supported by the National Natural Science Foundation of China (60673139, 60473073, 60573090)

Biography: KOU Yue(1980-), female, Ph.D. candidate, research direction: Deep Web, query processing. E-mail: kouyue@ise.neu.edu.cn

† To whom correspondence should be addressed. E-mail: yuge@ise.neu.edu.cn

0 Introduction

By applying Deep Web crawling, useful information can be integrated and returned[1]. Using previous query results to answer the current query can reduce the response time perceived by users. Hence caching is an important technique to enhance the query efficiency.

The application of cache has received increasing attention the last years. Florescu[2] exploits declarative Web site specifications for DB caching. Companies such as Akamai and Digital Island have provided services to cache static objects. Current application servers such as BEA WebLogic[3] and IBM Websphere[4] ease the development of Web sites through powerful components for HTML caching. However, customization of the Web site's runtime policy is less flexible, especially in the diversiform environment of Deep Web. Weave is a Web site management system that provides a language to specify a customized cache management strategy[5]. Oracle web cache uses a time-based invalidation mechanism by detecting changes on the back-end database and invalidating cached objects[6]. Instead of transferring the complete document the changes compared to some common base are computed, which reduces the cost of information transfer[7]. But for Deep Web where the information is more dynamic and massive, it is difficult for these techniques to provide better performance in dynamic maintenance because they directly consider the large scale of results as cached data.

Before getting into technical details, we start by describing the motivating examples for our work. Suppose Q_1 "find all papers containing the keywords 'database' and 'principle'" will be cached and the contents of each returned record is an object in format of PDF file. However, directly caching these large-scale contents can lead to some problems: First, storing these contents as cached data will result in excessive storage cost. Second, due to dynamic nature of Web data, results of previous queries stored in cache are possibly outdated at once[8]. Thus the returned records should be reorganized into an effective structure before caching. Suppose a subsequent query Q_2 "find all papers containing the keywords 'DB' and 'principle'" does not hit the cache. Then Q_2 is likely to be added into cache as a new record. But 'DB' is equivalent with 'database' in semantics. Rather than cache the similar data repeatedly, we should cluster them together. Thus our motivation lies in a new cache structure suitable for Deep Web should be defined. Some management and optimization measures should be performed to improve cache's performances.

In this paper we focus on providing a cache mechanism based on Top-*K* data source for Deep Web query. We begin by the follow assumption: Given a set of keywords, a result set from different data sources has been acquired. The contributions in this paper can be summarized as follows. Through analyzing feasibilities, a new effective cache structure is defined. By integrating techniques from IR and Top-*K*, a data reorganization strategy is presented to model KDS-CM. Some effective measures about cache management and optimization are also proposed. An experimental study is proposed to determine the effectiveness of KDS-CM in execution cost and dynamic maintenance.

1 Overview of KDS-CM

As it can be seen in Fig.1, KDS-CM includes two parts: cached data reorganization and cache management and optimization.

1.1 Cached Data Reorganization

For Deep Web, integrated query results are not fit for serving as cached data due to their expensive space cost and dynamic nature. So the returned records from Deep Web should be reorganized into an effective structure.

Definition 1 Cache structure: Cache is essentially a hash table where an entry consists a (key, value) pair. The key is the query description $\{k_1,\cdots, k_n\}$. The value is a set of answers represented as $\{\langle d_1, \text{score}_1\rangle, \cdots, \langle d_k, \text{score}_k\rangle\}$. It means the Top-*K* data source each with a matching score is stored as cached data.

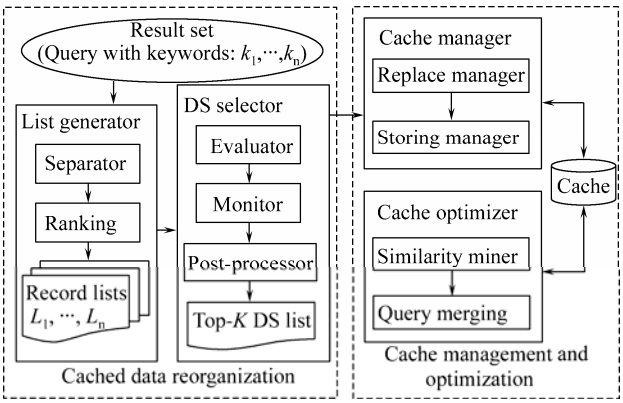

Fig.1 Overview of KDS-CM

The original ranked result set is from different data sources. We have known the ranked order of records in result set but not have identified the order of data sources. Our goal is to select the most representative data sources to replace the result set as cached data. Cached data reorganization contains the following components and information.

Given a query and its result set from different data sources, List Generator is used to generate some lists as bases for subsequent operations. In detail, Separator separates the query into some sub-queries, each of which corresponds to a keyword. Via tf×idf function defined in IR, the matching degree of result records with each sub-query is evaluated and ranked by Ranking. For each sub-query, a ranked Record List is generated, with its entries denoted as $\langle r, r\_\text{score}\rangle$, which represents the record *r* is one of the answers to this sub-query with the matching score of *r*_score. Data source (DS) selector is used to measure the matching degree between data sources and the given query. For each sub-query, the possible maximal and minimum score of data source will be evaluated by evaluator to aggregate the data source's final score. Monitor makes the selection terminate as early as possible. Post-processor is used to prune and refine the candidates to identify the final Top-*K* DS list.

1.2 Cache Management and Optimization

By performing some cache management and optimization measures including replacement strategy and clustering method, the efficiency of Deep Web query can be enhanced and the data redundancy in cache can be reduced effectively too. The components of this part will be introduced as follows.

Given a query, cache manager is responsible for the management of replacement strategies and storage mode. Replacement Manager performs a new replacement strategy based on LRU, which emphasizes on two factors: access frequency and matching precision. Storing manager is used to organize the cached data into a suitable storage mode.

Due to cached data redundancy, some optimization measures are performed by cache optimizer. Similarity miner calculates the similarities between the current query and an existing cached record. If the similarity exceeds a specified threshold, the new query is considered as redundant data and need to be merged with the existing one by query merging.

2 Data Reorganization in Cache

In order to transform the original result set to data sources, the answers should be reorganized to agree with the cache structure as Definition 1. IR and Top-*K* techniques are applied during data reorganization to identify the best data sources as early as possible.

2.1 Scoring Functions

The score of a record or data source is used to determine how closely it matches the target keyword set. Two kinds of scoring functions are defined.

Definition 2 Record score: Given a *n*-keywords query $\{k_1, \cdots, k_n\}$ and a record set $\{r_1, \cdots, r_m\}$, record score function denoted as $r\_score(r_j, k_i)$ equals $tf_{i,j} \times \log(m/df_i)$, which is based on tf×idf function in IR.

Definition 3 Data source score: Aiming at each sub-query, data source score $d\_sub\_score(d, k_i)$ is aggregated by the scores of records provided by *d*. Also for the entire query, the final data source score $d\_agg\_score(d)$ is aggregated by these $d\_sub\_score(d, k_i)$, $i=1,\cdots, n$ (see Eq.(1)).

$$d\_agg\_score(d) = \sum_i d\_sub\_score(d, k_i) = \sum_i \sum_j r\_score(r_j, k_i) \quad (1)$$

Due to the hugeness of result set, it is difficult to identify data sources' precise scores. Thus intervals are used to evaluate data sources' scores. With the increase of considered records in the process of data source selection, data sources' score intervals with lower and upper bounds are tightened and refined gradually.

The lower bound $sub\_L(d, k_i)$ of $d\_sub\_score(d, k_i)$ is *d*'s score aggregated by current seen records for k_i. Furthermore, the sum of these $sub\_L(d,k_i)$ is the lower bound of $d\_agg\_score(d)$ denoted as $agg\_L(d)$. The upper bound of $d\_sub\_score(d,k_i)$ is the maximum score that *d* might reach for k_i, which is denoted as $sub\_U(d,k_i)$. For each k_i, the maximum number of unseen records provided by *d* should be evaluated as $N_{unseen}(d,k_i)$ (Eq.(2)). Count(*r*) means the number of records from a data source in result set. For each k_i, $sub\_U(d, k_i)$ can be evaluated by Eq.(3) in which r_{new} is the current record retrieved from L_i. Then $agg\_U(d)$ is aggregated by them (see Eq.(4)).

$$N_{unseen}(d,k_i) \approx \max\{Count(r)\} - N_{seen}(d,k_i) \quad (2)$$

$$sub\_U(d,k_i) = sub\_L(d,k_i) + N_{unseen}(d,k_i) \times r\_score(r_{new}, k_i) \quad (3)$$

$$agg\_U(d) = \sum_i sub\_U(d,k_i) \quad (4)$$

2.2 Top-*K* Data Source Selection

The efficiency of Top-*K* data source selection relies on using intermediate answer scores to select relevant matches as early as possible. The Top-*K* algorithm includes two phases: evaluation and post-process.

2.2.1 Evaluation phase

The process of evaluation is an *n*-dimensional Top-*K* selection by scanning *n* ranked record lists. Via scanning the lists, the current data sources' scores are calculated and the candidates are collected. Figure 2 shows the algorithm of candidate set generation.

```
Input: ResultSet, Q = {k1, ···, kn}, K
Output: Candidate data sources
Method:
DSS={}      // Store the data sources retrieved currently
L={Li | Li is generated by List Generator , i=1, ···, n }
for ( pos=0; pos <= max{ Li.size | i=1, ···, n }; pos++ )
   for each Li ∈L do
      d_current = getDsByPos(Li, pos) //Get current data source from Li
      if ( d_current != null & d_current ∉ DSS ) then
         DSS = DSS ∪ {d_current } //Add a new DS to DSS
      end if
   end for
   updateScores(DSS) //Update sub_L, sub_U, agg_L and agg_U of DS
   SortedList = sort(DSS) //Sort DSS in descending order for agg_L
   min-k = min{agg_L(d) | d∈Top-K} //Get the K'th agg_L
   CAND={} // Store the candidate DS containing Top-K set
   for ( i=0; i <DSS.size; i++ )
      if (agg_U(SortedList.get(i)) >= min-k or DSS.size<=K) then
         CAND = CAND ∪SortedList.get(i)
      else return CAND //No unseen DS can qualify for the final Top-K
      end if
   end for
end for
return DSS //DS in DSS are all candidate ones
```

Fig.2 The algorithm of candidate set generation

Step 1 For each record list, retrieve a record pointed by the current cursor pos.

Step 2 Identify the relative data sources based on these current records.

Step 3 As for each data source generated from

lists, calculate sub_$L(d,k_i)$ and sub_$U(d,k_i)$ via the current state including $N_{\text{unseen}}(d,k_i)$ and r_score(r_{new}, k_i).

Step 4 As for all d, calculate the lower and upper bounds and sort them in a descending order.

Step 5 If the termination condition is satisfied, the current data sources will be returned as candidate set. Otherwise, move pos to next record and turn to Step 1.

2.2.2 Post-process phase

In this phase, data sources in candidate set should be further selected as the final Top-K based on their precise scores but not on their intervals. In general, the precise scores of all candidates should be calculated. In this paper, not all candidates but the minimum possible ones from them need to be computed precisely.

As a boundary, min-K divides the candidate set into two sections: Temporary Top-K (TK) and temporary Non-Top-K (TNK). TK contains the candidates whose agg_$L(d)$ are not less than min-K. Other data sources in candidate set constitute TNK. Firstly the intervals of candidates together should be projected. Secondly, for each data source in TK, identify whether there are interval intersections with TNK. If so, their scores need to be calculated precisely to select one from them with the highest score to final Top-K set. Otherwise, prune the data source from TK to the final Top-K set.

In previous example, candidate data sources' intervals are projected as shown in Fig.3. TK contains d_2, d_3 and d_1. TNK contains d_4. There are no data sources in TNK having intersections with them, so d_2 and d_3 must be in final Top-K. For d_1 and d_4, precise calculation should be performed furthermore.

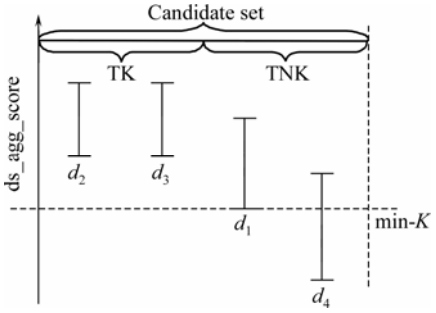

Fig. 3 Post-process performed on candidate set

3 Cache Management and Optimization

3.1 Cache Replacement Strategy

Cache management is critical to the performance in terms of access latency and cache hits. The worthiness of cached data can be measured by two factors: access frequency and matching precision. LRU[9,10] strategy only considers access frequency. But the matching precision is also important acting on cache worthiness.

The scores calculated during data source selection can reflect uses' preferences. The higher scores, the more precision of matching will be. By integrating access frequency and matching precision, some extensions based on LRU strategy are performed. The basic idea is as follows. Cached data which is the least recently used is evicted superiorly. When there are some cached data with the same access frequency, the one with least agg_L is evicted superiorly.

3.2 Cache Cluster

We can calculate the similarities of answers to estimate the relations of queries further. As for similar queries, they should be clustered together to eliminate the redundancy of cache.

Firstly, the new query result will be represented as N in the format of 〈agg_$L(d_1)$,⋯,agg_$L(d_k)$ 〉. Given an arbitrary existing cached record, its answer is represented as O. Note that elements in O must preserve the same order as N. Secondly, N and O are extended to have the same number of elements by filling scores 0. And then the similarity distance between N and O is calculated based on Eq.(5):

$$\text{Sim}(N,O)=\sigma(N,O)=\sum_{i=1}^{k}(n_i\times o_i)\Big/\left(\sqrt{\sum_{i=1}^{k}n_i^2}\times\sqrt{\sum_{i=1}^{k}o_i^2}\right) \quad (5)$$

For example, if the answers to the new query (with the keywords 'DB' and 'Principle') and an existing cached query (with the keywords 'Database' and 'Principle') are {〈d_1, 1.4〉, 〈d_2, 0.3〉, 〈d_3, 1.8〉} and {〈d_1, 1.4〉, 〈d_3, 1.9〉, 〈d_4, 0.5〉}. The final cosine similarity between them is 96.96%.

4 Experimental Evaluation

We use 10^4 records as result set generated by WISE-Integrator[11]. All experiments were run on a Dell machine with a 2.4 GHz P4, 512 MB of memory, and an 80 GB disk running Win2000.

Comparison with different reorganization strategies: The execution time of three different data reorganization strategies are compared: without early termination strategy performed (NET), with early termination strategy performed (ET) and with pruning based on ET performed (ET&P). Figure 4 shows the results for different

scales of data set. Merely evaluating partial records in result set, ET leads to significant reduction of execution cost. Furthermore, ET&P only calculates the minimum possible number of candidates precisely which results in further reduction of execution cost (See Fig.5).

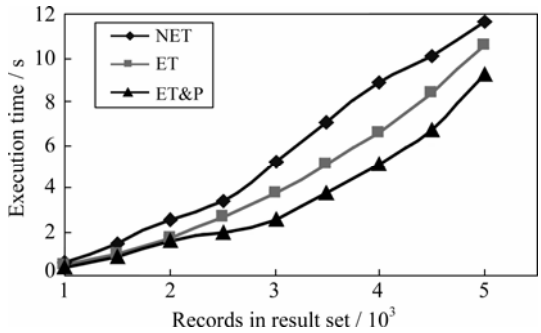

Fig.4 Different data reorganization implementations

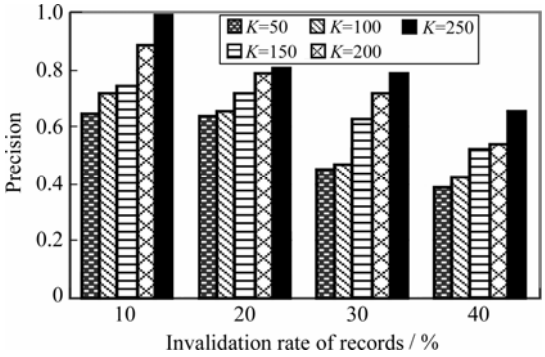

Fig.5 Stability of cached data

Stability of cached data: we randomly invalidate some records in result set with different rates. Based on the valid records a new Top-*K* set (denoted as TopK_{new}) can be obtained, which is considered as the current standard answers. By comparing the Top-*K* set stored in cache (denoted as TopK_{cache}) with the current standard answers, the query precision can be calculated. Figure 5 shows query precision is better even when the invalidation rate reaches 40%, because the invalidation of records can not easily defeat the Top-*K* set in cache.

$$\text{Precision} = \frac{|\{r|\text{provided by } d \text{ in Top } K_{cache}\} \cap \{r \mid \text{provided by } d \text{ in Top } K_{new}\}|}{|\{r \mid \text{provided by } d \text{ in Top } K_{cache}\}|}$$

5 Conclusion

In this paper we develop a research into cache modeling for Deep Web query. Currently accomplished work comprises: By integrating techniques from IR and Top-*K*, an effective cache structure is defined and data reorganization strategy is presented. Some management and optimization measures are also proposed.

Simulated experiments indicate based on early termination strategy and post-processing, the cache mechanism can be constructed with a lower execution cost. By performing data reorganization, the stability of cached data is improved effectively. Next we will make a further research on the efficiency of cache replacement and the intelligence of caching in semantics.

References

[1] Fetterly D, Manasse M, Najork M, *et al*. A Large-Scale Study of the Evolution of Web Pages[C] //*Proc of the* 12*th Intl WWW*. New York: ACM Press, 2004: 669.

[2] Florescu D, Levy A, Suciu D, *et al*. Optimization of Run Time Management of Data Intensive Web Sites[C] //*Proc of very Large DataBase*. Edinburgh: Morgan Kaufmann Publishers, 1999: 627.

[3] Karl F. BEA WebLogic Portal Documentation: Implementing User Profiles[EB/OL].[2003-05-16]. *http://edocs.bea.com/wlp/docs*70*/dev/usrgrp.htm#*998993.

[4] Bakalova R. WebSphere Dynamic Cache: Improving WebSphere Performance [EB/OL]. [2004-02-13]. *http://researchweb. watson.ibm.com/journal/sj/*432*/bakalova.pdf*.

[5] Florescu D, Yagoub K, Valduriez P, *et al*. WEAVE: A Data-Intensive Web Site Management System[EB/OL]. [2000-06-08].*http://wwwcaravel.inria.fr/dataFiles*.

[6] Anton J, Jacobs L, Liu X. Web Caching for Database Applications with Oracle Web Cache[C] //*Proc of SIGMOD*. New York: ACM Press, 2002: 594.

[7] Mogul J C, Douglis F, Feldmann A, *et al*. Potential Benefits of Delta Encoding and Data Compression for HTTP[C] //*Proc of SIGCOMM*. New York: ACM Press, 1997: 181.

[8] Yagoub K, Florescu D, Issarny V, *et al*. Caching Strategies for Data-Intensive Web Sites [C]//*Proc of very Large DataBase*. Berlin: Springer-Verlag, 2000: 188.

[9] Megiddo N, Modha D. Outperforming LRU with an Adaptive Replacement Cache[C] //*Proc of IEEE Computer*. New York: IEEE Press, 2004:58.

[10] Cheng-Yue Y C, Chen M S. Web Cache Replacement by Integrating Caching and Prefetching[C] //*Proc of CIKM*. McLean: Association for Computing Machinery Press, 2002:632.

[11] He H, Meng W Y, Yu C, *et al*. WISE-Integrator: A System for Extracting and Integrating Complex Web Search Interfaces of the Deep Web[C] //*Proc of very Large DataBase*. New York: Association for Computing Machinery Press , 2005: 1314.

□