# Learning from User's Behavior – A Technique for Personalized News Search Supporting Result Caching

Christos Bouras, Vassilis Poulopoulos, Panayiotis Silintziris
*Computer Engineers & Informatics Department*
*University of Patras, Greece*

## 1 Introduction

The technological advances in the World Wide Web, the low cost and the ease of access to it from any place in the world by using, not only conventional computers but also portable devices and cellular phones with advanced networking capabilities, has dramatically changed the way people face the need for information retrieval. More and more users migrate from traditional mass media to more interactive digital solutions such as Internet news portals. These digital neighborhoods provide a direct link to fresh and unfiltered stream of data, giving their subscribers the opportunity to stay up to date, in real time, with news and all kinds of information regarding their personal interests from all over the globe. This increasing popularity of Internet, as a vast digital data pool, which grows in an exponential rate, combined with the rather static and unchanged nature of human vocabularies, does not come without problems: Over time, it becomes a tedious task for the average user to successfully select a proper set of keywords that best describe his question and then locate the "right" piece of information in an ocean of irrelevant data. Also, the computational and memory load in the servers, which run the search engines that provide users with results, will probably increase at a comparable, if not higher, rate in the years to come.

Under these circumstances, cleverer search engines should be deployed on most news portals to help users find what they are looking for with less effort. In the majority of the currently existing search engines, when different users submit the same query, the same results are returned in the same order, regardless of who submitted the query. Obviously, it is unlikely that all the users of a search engine are so similar in their demands that a sole approach to searching fits all needs. Indeed, in terms of searching, one half of all retrieved documents have been reported to be irrelevant compared to what the user expected (Casaola, 1998). Additionally, a number of studies have shown that a vast majority of queries to search engines are short and underspecified (Jansen et al., 2000) and different users may have completely different intentions for the same query (Lawrence, 2000, Krovetz & Croft, 1992). The explanation is simple as one keyword or a limited set of keywords cannot always be an unambiguous guide to determine what a user is exactly interested in. This is the point where the personalized search can be of essential help. Presumably, information retrieval will be more efficient if individual users' idiosyncrasies are taken into account. Such a search strategy could decide autonomously for each user whether he is interested in an article and, in the opposite case, prevent it from being displayed. By modeling the user appropriately and personalize search according to his individual demands, we can achieve an improvement in the retrieval accuracy.

Throughout literature, we can identify two major directions for search personalization. *Query expansion* and *Result Processing* can complement each other and by understanding both the user and the context, a

breakthrough in search efficiency can be achieved (Lawrence, 2000, Xu & Croft, 1996). Query expansion is a way of solving the problem of word mismatch, which arises when users employ different terms than those used by content authors, to describe the same concept. This is achieved by expanding submitted queries with more relative words or phrases and it may offer an efficient solution (Xu & Croft, 1996). Context can take different forms, like category manually selected by the user (Glover et al., 1999) or combination of titles and descriptions of clicked search results after an initial query has been submitted (Leoryet al., 2003). On the other hand, result processing includes filtering and reorganizing of the search results in order to provide the user with a more refined output. This filtering can be either in the domain of the returned results (Oyama et al., 2004), eliminating in this way documents irrelevant to specific web domains, or in the news items that may not be of interest to a given user, according to that user's explicit (through rankings) or implicit (viewing and order duration) feedback, leading in personalized views of the result. Another approach to result processing deals with re-organization and re-ranking of the results, which is one of the major ideas of our work. This may involve the construction of a user profile over time with resources such as issued queries and visited links as in (Teevan et al., 2005). Previous queries and summaries of clicked results could also be used for re-ranking in the current session as in (Shen et al., 2005(1)). In addition to these server-side techniques, some client-side techniques have been proposed in the past, as in (Shen et al., 2005(2)), where query expansion and result re-ranking are employed on the basis of the immediately preceding query and of summaries of viewed results.

In our work, we use a enhanced combination of query expansion and results processing to produce personalized output. The difference with the aforementioned implementations, regarding the query expansion, is that it is executed after the first unranked set of result has been retrieved, participating and contributing in this way in the result processing. In the procedure, which we will describe, we expand the user's query with keywords that the engine has selected for a category or for a user. This selection is based on the user's previous search sessions and the categorization system of peRSSonal. By assigning increased or decreased importance weights to some keywords, it becomes feasible for the engine to obtain some kind of knowledge over the user's preferences. Thus, for each resulting article, a relevance factor is computed from keywords weight allowing the re-ranking of articles according to it.

We also explore the possibilities of speeding up the operation of the search engine of peRSSonal[1]. peRSSonal is a web-based mechanism for the retrieval, processing and presentation in a personalized view of articles and RSS feeds collected from major Internet News portals. In traditional search engines, when each query is submitted, a new database index search procedure is initiated to find and return a number of related articles (with a short summary for each one), which are relative to the given query. Depending on the complexity of the query, the engine potentially makes several accesses to the database data and metadata, probably residing in a secondary storage, and may consume significant computer resources. If a query is popular among different users or frequently used by the same user, then caching its results may improve performance significantly by reducing the computation and I/O overhead of the query evaluation, since the query results will be calculated only once: when the query is submitted for the first time.

The technique of caching query results and documents as a way to reduce access latency is being extensively used on the Web since its very first days. A simple client-side solution is to cache search results from the web browser inside the computer's main memory. This solution, although being widely used, produces poor results as it rarely leads to high hit rates (Abrams et al., 1995). As an improvement, caching proxy servers are employed. The large caches of the proxies serve a stream of requests coming from a large number of clients. However, because of the fact that even these large cache memories eventually fill up, there have been developed cache replacement policies. One of the first replacement policies considered is the LRU (least recently used) algorithm and its variations. LRU is based on the heuristic that the documents not used recently are probably not to be requested in the near future. One first extension of LRU uses heuristics that give priority to the caching of small documents instead of large documents (Markatos, 1996). A straighter alternative is the complete removal of large documents (Williams et al, 1996, Pitkow & Recker, 1994). The main idea

[1] http://perssonal.cti.gr

behind these variations originates to statistics showing that smaller documents were accessed more frequently than larger ones. Some other policies (Scheuearmann et al., 1997) also take into account network latency in order to avoid replacing documents that take a lot of time to download. By combining all above improvements, some techniques (Cao & Irani, 1997, Lorenzetti et al., 1998) aggregate all the aforementioned factors (access recency, size and latency) into a weight and try to keep in the cache the documents with the largest values.

In (Spink et al., 1998) and (Jansen et al., 1998) we can find and analysis over the transaction logs posed by users of Excite, a major Internet search engine. Among their most interesting findings, regarding how users search the web and what they search on it, are that users tend to ask short queries and choose to view no more than 2-3 pages of the query's results. (Silverstein et al., 1998) have studied a very large log of AltaVista queries and among other results they report that the average frequency of the queries in the trace was 3.97. That is, each query was submitted four times, which implies that caching search queries' results may lead to high hit rates.

These findings are particularly encouraging for our approach as they demonstrate that there is a capable amount of locality in search engine queries. In our approach, we take advantage of this space and time locality, and we cache the results from very recently used queries (RSS feeds tend become older much faster than URLs coming from web search engines) in order to reduce the latency on the client side and the database-processing load from the server side. Because of the fact that the caching is server side, both registered and unregistered users of the portal can take benefit.

## 2 Architecture

The architecture of the system is distributed and based on standalone subsystems but the procedure to reach at the desired result is actually sequential, meaning by this that the data flow is representative of the subsystems of which the mechanism consists. Another noticeable architectural characteristic is the existence of modularity throughout the system's lines. This section is a description of how these features are integrated into the mechanism. We are putting the focus on the personalized search subsystem, though brief analysis of the other modules is presented in order to cross-connect the features of our system. As already mentioned, the personalized search procedure both produces and consumes knowledge from the personalization mechanism in order to enhanced output quality.

The architectural schema consists of a series of subsystems, as depicted in Figure 1. The collaboration between the distributed parts relies on the open standards for input and output that are supported by each part of the system and on the communication with a centralized database.

The general procedure of the mechanism is as follows: at first, web pages are captured and only the useful text (drop stop words, punctuation etc.) is extracted from them. Then, the extracted text is parsed followed by summarization and categorization. Finally we have the presentation of the personalized results to the end user.

For the first step, a simple web crawler is deployed, which uses as input the addresses extracted from the RSS feeds. Theses feeds contain the web links to the sites where the articles exist. The crawler fetches only the html page, without elements such as referenced images, videos, CSS (Cascading Style Sheets) or JavaScript files. Thus, the database is filled with pages ready for input to the 1st level of analysis, during which, the system isolates the "useful" text from the html source. Useful text contains the article's title and main body. This procedure can be found in (Bouras et al, 2005).

In the 2nd level of analysis, XML files containing the title and the body of articles are received as input, targeting at applying pre-processing algorithms on this text in order to provide as output the keywords, their location in the text together with their absolute frequency in it (number of times met in the text). These results are the primary input to the 3rd level of analysis. In the 3rd level of analysis, the summarization and categorization technique takes place. Its main scope is to characterize the articles with a label (category) and come up with a summary of them. Details about these procedures can be found in (Bouras et al., 2006). The core of our work can be found in the 4th level of analysis, where the results are presented to the end user in a personalized view. The algorithms of personalization, which will be analyzed in the next section, take as input,

information about the user's profile and preferences, collected and processed during his past sessions in the portal. For each user, a set of keywords with assigned relevance weights is used as the main criteria for the final order in which the articles will be presented as well any extra articles that the engine considers as "possibly interesting" for the specific user, although these articles might not be directly linked to the specified query. This way the system manages to enrich the user's experience.

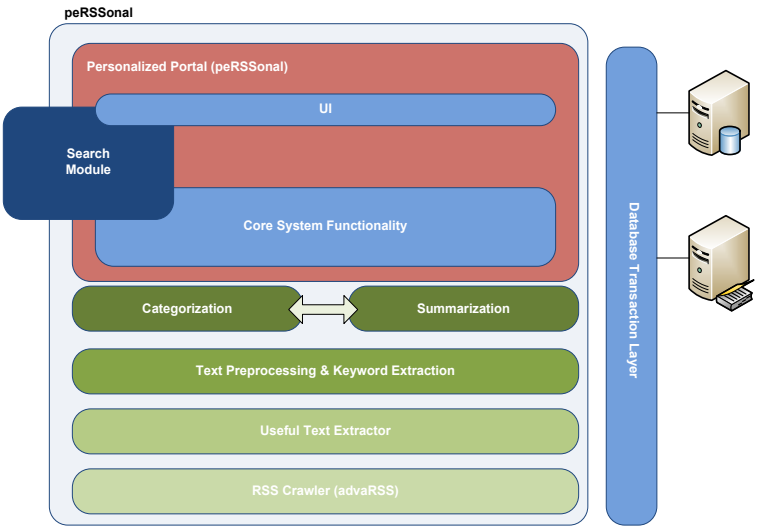


**Figure 1:** The search module within PeRSSonal's Architecture

In Figure 2, we can see the general schema and flow of the advanced and personalized search sub-module. The user submits the keywords together with the search configuration options through a form. The keywords first pass through a Stemmer so that they get in the form that they are stored in the database. A caching algorithm is used to search for similar queries submitted in the past from the same user, and if matches are found, cached results are directly obtained improving in this way the search speed. In the next step, the query is enriched with more keywords, relative to the user's profile acting as a base for more relevant articles to be retrieved. In the final step, a weighting algorithm is deployed to adjust the weights of the keywords that the engine has characterized as favorite for the user who conducts the query, according to his past searches and general behavior in the system. We will focalize on this module of the system as this is the one that implements and executes the personalized search procedure of the system.

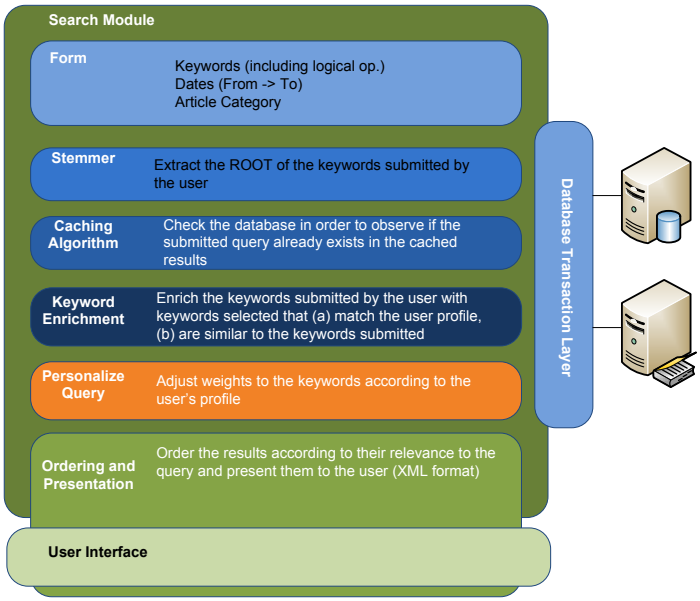


**Figure 2:** Search Module Internal Architecture

# 3 Algorithmic Aspects

In this section we will describe the algorithmic aspects of the proposed mechanism.

## 3.1 Configuration and Keyword Refinement

Before the engine triggers the search procedure, the user has first to configure the search. Apart from the specified keywords, a few other options are provided, including the date period for the target result, the selection of the logical operation ("OR" and "AND), which will be performed in the articles matching and the thematic category of the desired output. Before proceeding with query search operation, the engine passes the keywords through a stemmer, which implements the Porter Stemming Algorithm on the English language. Thus, we enable the integration of the search engine with the rest of system, which is build on stems rather than full words for the articles categorization. Additionally, simple duplicates elimination is executed on the stems, taking into account that some duplicates are not removed due to their nature of lexicological co-existence.

## 3.2 Results Fetching

In the first phase of the algorithm, we fetch from the database the articles that constitute a direct match to the submitted query, taking into account the configuration of the search options. We should emphasize here, that if the user has requested articles of a particular category, then an article is fetched only if it has the biggest frequency in the specified category among all categories. This can be achieved straightforward, as peRSSonal maintains the frequency per category for each article it collects from the Internet, by associating the frequency of its keywords with every thematic category. In the algorithm we can see the execution of the above procedure.

```
Keywords[] = Stem_Query();
Query = Construct_SQL(Keywords[], Category, Dates, Logical_Operator);
Articles[] = Fetch_Articles(Query);
Foreach(Ar as Articles[])
      If(Is_Selected(Category)) Then
            High_Freq_Cat = Ar.Find_Category_With_Biggest_Frequency();
            If(Category==High_Freq_Cat) Then
                  Ar.Store_Article();
            End If
      Else
            Ar.Store_Article();
      End If
End For
```

The results are temporarily stored in array structures so that they can be sorted in later stages of the algorithm execution.

### 3.3 Query Expansion and Personalization on User

In the next phase of the search procedure, we have to refine the order of the articles to be shown both for the case of a generic and for the case of a personalized search.

In the first case of a generic search of an anonymous user, the query is expanded with more keywords so that results with highest relevance to the search request can be obtained. For this reason the algorithm presented below is used. This algorithm takes each keyword in the query and assigns a weight to it. Keyword weights in our experiments start from 0.1 for the first keyword in the query, with each next keyword having weight 0.01 less than the previous one. Furthermore, we compute its absolute frequency for all categories in the system. The purpose is to expand the query of an anonymous user by enriching it with more keywords from categories with high relevance to the query's original keywords. To accomplish it, we use in our experiment different values for the representativity factor ranging from 1.5 to 4.5. The way the representativity factor affects the number of the added keywords is described later in the chapter. For our system, a representativity factor of value *R* means that a keyword *K* is considered as representative of category *C*, only if its absolute frequency of appearance in *C* is *R* times higher compared to its absolute frequency in the category where keyword K has the second highest absolute frequency. Consequently, higher values of *R* yield lower possibility for the user to select category-representative keywords, while lower values allow for easier query expansion with more keywords to refine the results. In case we find high relevance to a category, we proceed with expanding the query with more keywords from this category. For this reason, we retrieve the keywords having absolute frequency greater or equal to the particular keyword in the query and we select these two that have frequency just above or equal to the query's original keyword. Each new child keyword gets the weight of its higher weighted parent keyword (for keywords with more than two parents).

```
Keywords[] = Stem_Query();
Define_Relativity_Factor(R);
Enrich_Keywords[] = Empty_Array();
Foreach(Key as Keywords[])
      Parent_Weight = Assign_Weight();
      Frequencies[] = Find_Frequence_For_Each_Category(Key);
      Freq_1 = Max(Frequencies[]);
      Freq_2 = Second_Highest(Frequencies[]);
      If(Freq_1 > R * Freq_2)Then
            Child_Keys[] = Find_Two_More_Keywords();
```

```
            If(Child_Keys[] Not In Enrich_Keywords[])Then
                  Child_Keys_Weight = Parent_Weight;
                  Enrich_Keywords.add(Child_Keys[]);
            End If
      End If
End For
```

After the expansion process has been completed and the weights of all keywords have been adjusted, including the keywords added to expand the query, we can compute the relevance of an article to the user's query according to the following formula:

$$I_{relevance} = \frac{\sum_{i=1}^{N}(x_i y_i)}{\sqrt{\sum_{i=1}^{N}(x_i^2)}\sqrt{\sum_{i=1}^{M}(z_i^2)}}$$

In this formula, $x_i$ , $y_i$ denote the computed weight and the actual frequency of the i[th] keyword X in the expanded query of *N* keywords and the actual frequency of the i[th] keyword Y in the document respectively where X and Y are the common keywords, while $z_i$ denotes the actual frequency of the i[th] keyword of the list of all *M* keywords existing in the fetched article. Obviously, this index gives a normalized measure of an article's relevance to a given query.

In the case of a personalized search, the relevance of each article to the query has to be computed by taking into account the individual profile and the preferences of the user committing the search request. In the profile of each user in the database we have assigned different weights to a large group of keywords so that we can more efficiently describe the user preferences. Keywords with high relevance over a user's favorite thematic category gain positive weights, while keywords belonging in categories which are of less or of no interest to the user have lower or negative weights accordingly. The profile for each user is initialized during his registration into the system, where he can provide an index of interest (-5 to 5) for each thematic category. This profile can change dynamically overtime in accordance to the user's overall behavior inside the system (visited articles, time spent over an article etc.). Based on such profiles, we can compute for each article in the system an index of relevance for each user. The formula used for this reason in our implementation of the personalized search is the following:

$$I_{personalized} = \frac{\sum_{i=1}^{N}(p_i w_i)}{\sqrt{\sum_{i=1}^{N}(p_i^2)}\sqrt{\sum_{i=1}^{M}(z_i^2)}}$$

In the above formula, N is the number of keywords that have been assigned a weight (negative or positive) in the user's profile, $p_i$ is the weight of each such keyword, $w_i$ is the actual frequency of each such keyword in the specific article; M is the total number of keywords in the article and $z_i$ is their actual frequencies inside the article. From this formula we can see how highly weighted keywords in the user's profile can increase the relevance of an article, therefore the rank in which it will be finally displayed, and that for different users we obtain different relevance for a given article. For example, if a user has declared high interest over education and no interest over sports, then keywords such as "school", "pupil", "teacher", etc. have obtained high weight leading in high computed relevance for articles around education, while keywords such as "football", "athlete", etc. have negative weights, leading in low relevance for articles around sports.

Finally, by sorting the fetched articles array according to their computed relevance, we manage to present the user with results personalized and targeted to his profile and preferences.

### 3.4 Caching Algorithm

In this section, we shall analyze the caching algorithm of our search implementation, that is used in order to enhance the speed of the complete procedure and lessen the computational overload of our resources.
Prior to searching for the result articles, the system searches for cached data from previous search sessions. All cached data are stored on the server's storage space and the caching algorithm also operates in the server's memory so the procedure, which will be described, will be of benefit for both registered users (members) as well as unregistered users (guests) of the portal without creating any computational overhead for their machines.

For each submitted query, we store in a separate table in our database information about how it was configured. This information includes the id of the user who made the search request, the exact time of the request (as a timestamp), the keywords used in the query formatted in a comma-separated list and a string containing information about the desired category of the results and the logical operation, which was selected for the matching. For the above data, our caching algorithm operates in a static manner. For example, if a user submits a query containing the keywords "nuclear technology", by selecting the "science" category as the target category for the returned articles, this query will not match against an existing (cached) query which contains the same keywords but which was in the first case cached for results on the "politics" category. Also, when a query containing more than one keyword is submitted, it will not match against cached queries containing subsets or supersets of the keyword set of the submitted query. For example, if the incoming query contains the keywords "Monaco circuit formula", probably referring to the famous Grand Prix race, it will not be considered the same with a cached query containing the keywords "circuit formula" which probably refers to an electrical circuit formula of physics. This decision for the implementation was taken in order to avoid semantic ambiguities in the keywords matching process.

The dynamic logic of our caching algorithm lies in the target date intervals of a search request, which are represented by the "date from" and "date to" fields in the search configuration form of the portal. This perspective of caching was chosen after considering the fact that is very common for many web users to submit identical queries repeatedly in the same day or during a very short period of days. The algorithm designed for this reason takes into account the following 4 cases for the cached "date from" and "date to" fields and submitted "date from" and "date to" fields:

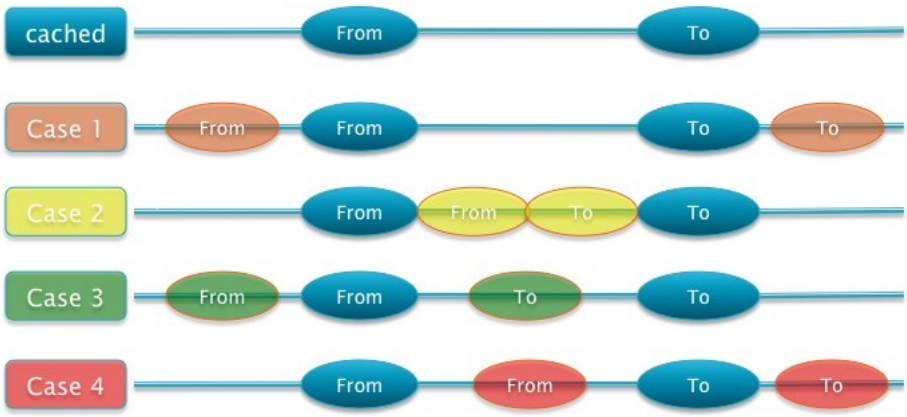


**Figure 3:** Date-Matching Scenarios

**1st Case**: the DATE FROM-TO interval of the submitted query is a subset of the DATE FROM-TO interval of the cached query. In this case, we have the whole set of the desired articles in our cache plus some

articles out of the requested data interval. The implementation fetches all the cached results and it filters out the articles, which were published before DATE FROM and these, which were published after DATE TO attribute of the submitted request. The server's cache is not updated with new articles because in this case no search is performed in the articles database.

**2nd Case**: the DATE FROM of the submitted query is before the DATE FROM of the cached query and the DATE TO of the submitted query is after the TO DATE TO of the cached query. In this case, the desired articles are a superset of the articles, which are cached in the database. As a consequence, the algorithm fetches all the cached results but it also performs a new search for articles in the date intervals before and after the cached date interval. When the search procedure finishes, the algorithm updates the cache by extending it to include the new articles and by changing the DATE FROM and DATE TO attributes so that they can be properly used for future searches.

**3rd Case**: the DATE FROM of the submitted query is before the DATE FROM of the cached query and the DATE TO of the submitted query is between the DATE FROM and DATE TO of the cached query. In this case, a portion of the desired articles exists in the cache. The algorithm first fetches all the results and then it filters out the articles, which are after the DATE TO date of the submitted request. Furthermore, a new search is initiated for articles not existing in the cache memory. For the new search the DATE FROM and DATE TO dates become the DATE FROM date of the submitted query and the DATE FROM date of the cached query.

**4th Case**: The form case is similar to the third case but in the opposite date direction. The final results consist of the cached results between DATE FROM date of the submitted request, the DATE TO date of the cached request and the new articles coming from a new search between the DATE TO date of the cached query and the DATE TO date of the submitted query.

We should notice that for the cached results data, an expiration mechanism is deployed. Every cached query is valid for a small number of days, in order to keep the engine's output to the end user as accurate and fresh as possible. Whenever a search for a matching with the cached results is performed, cached data that have expired are deleted from the database and are replaced with new one. It is also possible for the same query to exist in more than one cached records as long as they have not expired. The selection of the proper expiration time for the cached data will be discussed in the next paragraph. The overall operation of the algorithm is presented with pseudo code in the algorithm below.

```
match = Search_In_Cache(query);
If(Is_Found(match))
expired = Check_Expiration(match);
If(expired==true)
       Delete_From_Cache(match);
       results[] = Execute_New_Search(query);
       Insert_In_Cache(results[]);
Else
       Check_Case(1):
              results[] = Fetch_Results(match);
              results[] = Filter(results[]);
       Check_Case(2)
              results[] = Fetch_Results(match);
              results.append(Execute_New_Search_Before());
              results.append(Execute_New_Search_After());
              Update_Cache(results[]);
       Check_Case(3):
              results[] = Fetch_Results(match);
              results = Filter(results[]);
              results.append(Execute_New_Search_Before());
              Update_Cache(results[]);
       Check_Case(4):
```

```
            results[] = Fetch_Results(match);
            results = Filter(results[]);
            results.append(Execute_New_Search_After());
            Update_Cache(results[]);
Endif
Else
results[] = Execute_New_Search(query);
Insert_In_Cache(results[]);
Endif
```

By examining this algorithm, operating in the server's memory, we can see that in all four cases, we achieve to limit the computational overhead of a fresh search in the database by replacing it with some overhead for cached results filtering. However, this filtering is implemented with simple XML parsing routines and cannot be considered as a heavy task for the server. The most significant improvement happens in the first case, where no new search is performed and all the results are fetched directly from the cache. This is a great benefit to our method as this is the most common case, where the users submits the same query over and over without changing the DATE FROM and DATE TO fields or by shrinking the desired date borders. The worst case is the second, where the user expands his query in both time directions (before and after) in order to get more results in the output. In this case, the engine has to perform two new searches, followed by an update in the database cache. However, this is the rarest case, as the average user tends to shrink the date interval rather than expanding it, when he repeatedly submits an identical query in order to get more date-precise and date-focused results. In the other two situations, one new search is executed each time and one update is committed in the database. This means that in an average case, we can save more than 50% of our computational overhead when the expansion of the date borders (with the newly submitted query) are not bigger than the cached results date interval.

# 4 Experimental Evaluation

In this section we provide experimental evaluation that is done to the system. The experiments presented are limited to a number of virtual users as the complete system is still under experimental evaluation. Nevertheless, the results concerning the searching mechanism are promising.

## 4.1 Representativity Factor

In the query expansion process, we manage to enrich the user's query by using a number that we call representativity factor. This number, as described in previous paragraph, represents the user's possibility of selecting keywords that are highly representative of some thematic categories of the system. When the user picks these keywords in his query, the engine expands the query it by adding in it more keywords from the category with the highest relevance. In Figure 4, we present the relation between the representativity factor and the number of keywords in the system, which automatically become category representative. For the construction of this chart, we first obtained the 100 most representative keywords for 7 of the basic thematic categories existing in PeRSSonal. We removed the duplicates and we ended with 490 different keywords. As we can see from this graphic, the possibility of selecting category representative keywords is relatively high, almost 70%, when the representativity factor takes values below 2 and falls at 50% and lower for values above 3.5, making it more "difficult" to select representative keywords. The extra-added keywords do not participate in the articles result set, which will be presented to the user, but their role is to aid in the re-ranking procedure of the final results. We should notice that neither low values for the factor should be used, as this would make it very possible to re-rank the output articles based on keywords the user had not in mind when submitting the

query, nor low values as this would make the final articles result difficult to focalize one the thematic category, but it would spread among different categories.

### 4.2 Personalized Vs. Non-Personalized Search

In order to compare the results of a personalized search to the results of a non-personalized search, we conducted our experiments for three virtual users we created. During the registration process, we gave each user a positive preference bias over one category and negative bias for all other categories of the peRSSonal, simulating in this way different groups of people. In the tables and graphics that follow, we consider user A as a user with high preference over sport news, user B with high preference over business news and user C with high preference over technological news. This said, a user with favor over a particular category, is not excluded from being presented or selecting to view articles from different categories but the engine tries to adapt as much as possible the results to his profile. The queries we experimented on consist of category independent keywords so that the results we obtain are generic. Examples of such keywords are 'Sunday', 'New York', 'red', 'environment' etc. In this point we should notice that prior to gathering the presented results, we trained the system for each user separately with articles that were of high interest for these users. As a result, his profile became more category-polarized on the category the user initially indicated as categories of interest.

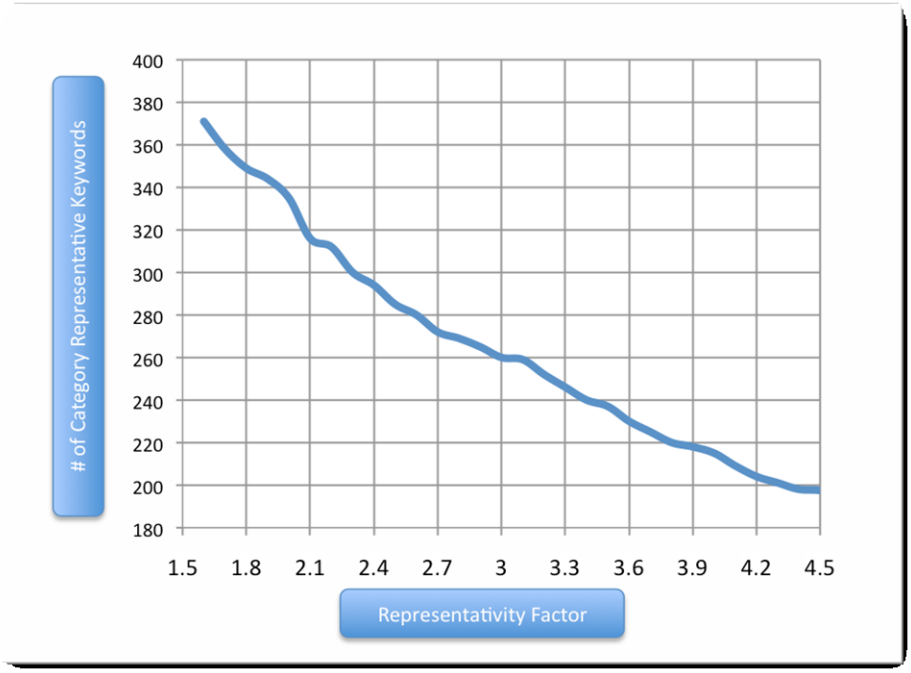


**Figure 4:** Query Expansion - Representativity Factor

In the first phase of the experiment, we conduct a non-personalized search (anonymous user) on a generic query. The first 60 articles we come up with are presented in table 1. For the sake of simplicity, we do not show the real article ids in the table, as they are stored in the database, but the rank in which they are returned from the search. This rank will be used for the evaluation of the personalized results for each user. We also present

the category for each article (1=Business, 2=Sports, 3=Health, 4=Technology, 5=Science, 6=Education and 7=Entertainment) as well as its relevance to the query, computed from the formulas of section 4.

As we can see from Table 1, because of the generic query the results are spread among the different categories of PeRSSonal allowing different users to find articles of interest. For this experiment the articles that our imaginary users selected to view are presented in the table 2.

| Art. Id | Category | Relevance | Art. Id | Category | Relevance | Art. Id | Category | Relevance |
|---|---|---|---|---|---|---|---|---|
| 1 | 4 | *0.836* | 21 | 4 | *0.597* | 41 | 2 | *0.340* |
| 2 | 1 | *0.822* | 22 | 1 | *0.562* | 42 | 3 | *0.332* |
| 3 | 1 | *0.807* | 23 | 2 | *0.559* | 43 | 3 | *0.284* |
| 4 | 4 | *0.782* | 24 | 3 | *0.556* | 44 | 4 | *0.284* |
| 5 | 2 | *0.766* | 25 | 2 | *0.545* | 45 | 1 | *0.269* |
| 6 | 5 | *0.762* | 26 | 2 | *0.533* | 46 | 6 | *0.262* |
| 7 | 3 | *0.760* | 27 | 5 | *0.532* | 47 | 1 | *0.213* |
| 8 | 2 | *0.743* | 28 | 1 | *0.530* | 48 | 2 | *0.199* |
| 9 | 3 | *0.735* | 29 | 6 | *0.528* | 49 | 7 | *0.178* |
| 10 | 4 | *0.732* | 30 | 5 | *0.525* | 50 | 4 | *0.166* |
| 11 | 6 | *0.707* | 31 | 3 | *0.519* | 51 | 1 | *0.156* |
| 12 | 1 | *0.706* | 32 | 4 | *0.515* | 52 | 2 | *0.117* |
| 13 | 6 | *0.702* | 33 | 2 | *0.487* | 53 | 1 | *0.111* |
| 14 | 3 | *0.632* | 34 | 6 | *0.476* | 54 | 1 | *0.091* |
| 15 | 2 | *0.624* | 35 | 2 | *0.474* | 55 | 3 | *0.085* |
| 16 | 2 | *0.622* | 36 | 1 | *0.468* | 56 | 3 | *0.067* |
| 17 | 4 | *0.617* | 37 | 2 | *0.427* | 57 | 1 | *0.065* |
| 18 | 7 | *0.609* | 38 | 3 | *0.411* | 58 | 5 | *0.056* |
| 19 | 1 | *0.604* | 39 | 3 | *0.410* | 59 | 5 | *0.033* |
| 20 | 2 | *0.602* | 40 | 1 | *0.369* | 60 | 7 | *0.008* |

**Table 1:** All articles fetched in the non-personalized search with category and relevance

| User | Article Ids per Category |
|---|---|
| A | Sports(8,20,33,48) Business(12,54) |
| B | Business(12,19,40,45,57) Entertainment(18) |
| C | Technology(31,32,44) Health(7) Science(27) Business(19) |

**Table 2: Articles the virtual users selected in the non-personalized search**

In the second phase of the experiment, we conduct a personalized search for each of the three users, using the same query as in the generic search. This time, we present for each user the first 15 articles of the search result. For the evaluation of the results in the tables 3-5 that follow, we use as article id, the initial rank of the article as this was in the non-personalized search. The articles with id 'New' are articles that were not in the table 1 and that now have been fetched in the top 15 ranks for a user as a high relevance was computed for them. The highlighted articles in tables 3-5 show the articles that the user had selected in the non-personalized query of table 1.

| Personalized Rank | Article Id | Personalized Relevance | Non-personalized Relevance | Category |
|---|---|---|---|---|
| 1 | 5 | *0.846* | *0.766* | Sports |
| 2 | 8 | *0.839* | *0.743* | Sports |
| 3 | 12 | *0.833* | *0.706* | Business |
| 4 | 14 | *0.824* | *0.632* | Health |
| 5 | 3 | *0.779* | *0.807* | Business |
| 6 | 33 | *0.776* | *0.487* | Sports |
| 7 | New | *0.763* | *0.000* | Sports |
| 8 | 39 | *0.757* | *0.410* | Health |
| 9 | 35 | *0.738* | *0.474* | Sports |
| 10 | 7 | *0.738* | *0.000* | Health |
| 11 | 28 | *0.735* | *0.530* | Business |
| 12 | New | *0.734* | *0.000* | Sports |
| 13 | 18 | *0.730* | *0.609* | Entertainment |
| 14 | 20 | *0.729* | *0.602* | Sports |
| 15 | 6 | *0.698* | *0.762* | Science |

**Table 3:** Personalized Search for User A

| Personalized Rank | Article Id | Personalized Relevance | Non-Personalized Relevance | Category |
|---|---|---|---|---|
| 1 | 22 | *0.901* | *0.562* | Business |
| 2 | 40 | *0.871* | *0.369* | Business |
| 3 | 21 | *0.824* | *0.597* | Technology |
| 4 | 35 | *0.801* | *0.474* | Sports |
| 5 | 12 | *0.796* | *0.706* | Business |
| 6 | 28 | *0.788* | *0.530* | Business |
| 7 | 1 | *0.751* | *0.836* | Technology |
| 8 | 18 | *0.723* | *0.609* | Entertainment |
| 9 | New | *0.717* | *0.000* | Business |
| 10 | New | *0.715* | *0.000* | Sports |
| 11 | 6 | *0.699* | *0.762* | Science |
| 12 | 18 | *0.680* | *0.609* | Entertainment |
| 13 | New | *0.658* | *0.000* | Technology |
| 14 | 45 | *0.649* | *0.269* | Business |
| 15 | New | *0.640* | *0.000* | Sports |

**Table 4:** Personalized Search for User B

| Personalized Rank | Article Id | Personalized Relevance | Non-personalized Relevance | Category |
|---|---|---|---|---|
| 1 | 21 | *0.928* | *0.597* | Technology |
| 2 | 7 | *0.894* | *0.760* | Health |
| 3 | New | *0.877* | *0.000* | Technology |
| 4 | 19 | *0.830* | *0.604* | Business |
| 5 | 44 | *0.821* | *0.284* | Technology |
| 6 | 49 | *0.819* | *0.178* | Entertainment |
| 7 | 17 | *0.782* | *0.617* | Business |
| 8 | 30 | *0.760* | *0.525* | Science |
| 9 | 32 | *0.754* | *0.515* | Technology |
| 10 | 56 | *0.753* | *0.067* | Health |
| 11 | 10 | *0.741* | *0.732* | Technology |
| 12 | 36 | *0.722* | *0.468* | Business |
| 13 | 31 | *0.716* | *0.836* | Technology |
| 14 | 3 | *0.691* | *0.807* | Technology |
| 15 | New | *0.680* | *0.000* | Technology |

**Table 5:** Personalized Search for User C

With a quick look in the tables above, we can see that most of the articles that the three virtual users had selected in the non-personalized search have now been fetched in higher ranks with higher relevance. For example, for user A, four of the six articles he had selected in the non-personalized search in the first 60 results have moved several places higher in the rank of the personalized search, while three of them are now in the first ten results. This can be seen in figure 5, where the non-personalized and personalized rank of the selected articles is presented graphically.

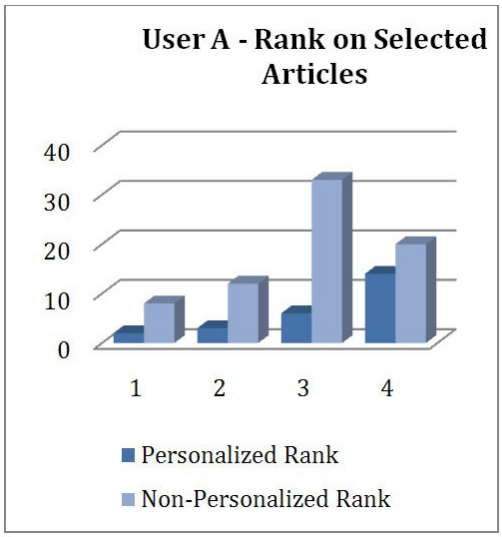


**Figure 5:** Rank on selected articles for User A

### 4.3 Evaluation of Caching

In our experiment to evaluate the caching algorithm, which was described in the previous paragraph, we create a virtual user to submit queries to the server. The executed queries consist of keywords from several thematic categories (sports, science, politics, etc.) used throughout the articles database of our system. We choose to test caching performance on queries containing no more than three keywords, in order for the output to contain a big number of articles and for the overall procedure to last as much as needed for our time measurements to be sufficient and capable of analysis and conclusions.

In this section, we will focus on the following issues:

- The performance of our server-side caching algorithm for the different scenarios of date matching against submitted queries.
- The way in which the number of cached documents affects the speed of caching and the storage space required on the server.
- How the selection of an expiration time for our cached queries results affects the quality and the accuracy of the output to the end user.

In the previous paragraph, we analyzed the way in which the algorithm tries to match a submitted query to find an identical cached record. During the experiment, we tested several queries, requesting articles from different categories, covering the period of the last six months. In the first phase, we used an empty cache memory and the server was configured to have the caching feature disabled. As it was expected, queries consisting of very focalized and specific keywords were processed very quickly. These queries are not of high interest concerning our analysis, as the number of articles containing such keywords are always quite limited and require small computational time to process.

The major problem exists with queries consisting of generic keywords, which can be found on a plethora of articles in the database. This class of queries makes heavier usage of system resources and can be considered as a good starting point to evaluate our method. In Figure 6, we can examine the results of caching on execution procedure speedup for three generic queries (‘sports’, ‘computers’, ‘health or body’), which returned over 5000 articles. This graphic depicts the time in seconds that the system needed to fetch the matching output from the database. The cases considered in this figure are cases 2, 3 and 4 of our algorithm, where only a subset of the results for the submitted query exists in the cache memory and the system will initiate a new search in the database to fetch articles for the missing date periods. The selection of the date period, for which the results were cached in the first place, before the actual queries were submitted, was a random number of days varying from 60 to 90. The actual query, which was to be evaluated, required articles published in the last 180 days. This means that the system had still to search for more articles than the number of articles it had already stored in its cache memory. In the results presented, we can notice that under some situations, the benefit reached almost half the time of the actual (without caching) time needed. As it was expected, the worst case is case 2, where two new un-cached searches have to be executed, one before and the other after the date period of the cached set. After that, we come up with three different sets of articles. Prior to presenting them to the end user, we have to re-sort them according to their degree of relevance to the initial query. For cases 3 and 4, the results are almost similar. The higher times in case 4 could be a consequence of a possibly high concentration of desired articles in the date period, for which the new search was initiated, combined with a reduced concentration of articles in the date period stored in the cache memory of the server.

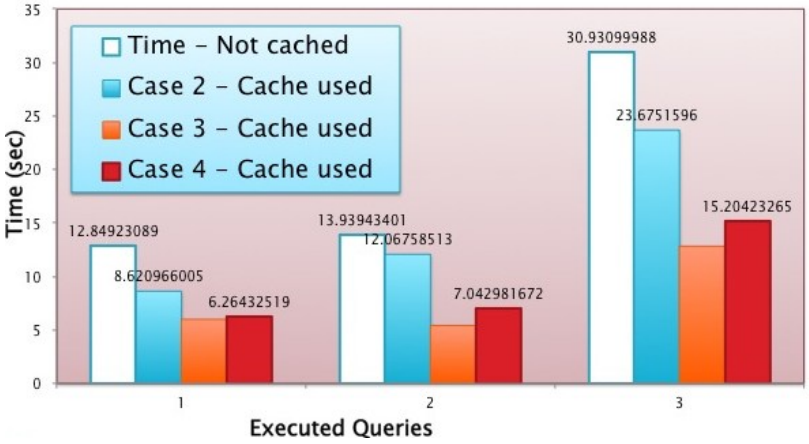


**Figure 6:** Time in seconds for un-cached searches and cached searches for cases 2,3,4

In the execution times measured throughout the experiment, an average 0.1 seconds were needed to fetch the articles from the cache memory, which is at average almost 3% of the overall time needed. Another 2% of the time was spent on re-sorting the two or three sets of results, according to their relevance to the query, in order to present them to the end user in the right order of relevance to the query. This said, it is expected for the case 1 of our algorithm to achieve an almost 95% speed up on the search. After the first execution of these queries, every next submission of the same request is serviced in under 0.1 seconds. Whenever results are cached for a query, every following identical one which demands articles inside the date period of the cached result, will be processed in almost zero time – only the time needed to fetch the results from the cache - no resorting is required in this case as we have only one already sorted set of articles. This reduces the computational overhead on the server for time demanding queries to the cost of the search procedure for only the first time they are executed. Every next time they are processed through the cache memory and the algorithm operating on it.

### 4.4 Cache Memory Size

Our second concern was to examine how the number of the cached articles per query in our cache, affects the overall algorithm performance and the size of the database table used to store the cached data. We executed a generic query for several numbers of cached articles by increasing each time the date period in which the caching occurred. The total number of articles for this query was 4782 over a period of 4 months. For this experiment, we tested cases 2,3 and 4 of our algorithm, so that in every submitted request, a part of the results were not contained in the cache and the engine could not rely only on the cached data to create the output.

From the graphical representation of Figure 7, relating the percentage of execution time speedup with the percentage of cached results on total results, we can notice that the search execution time reduces at an average 50% when a little less than 40% of the output has been cached. As the total number of articles in this test covered a period of four months, we can say, by statistic, that the 40% of the results would be retrieved by a search in a period of less than two months, which, speaking modestly, is a rather limited date interval on a common search. By that, it is meant that if a user submitted a search query, requesting articles for a period of more than two months, then every next time he submits an identical request, it would take at most half the time to be processed. If we add to this the fact that the algorithm updates the cache memory with new results, every time an extended (in terms of dates) version of an already cached query is submitted (the percentage of cached

results probably increases and never decreases in every search), we could get even more improved execution times.

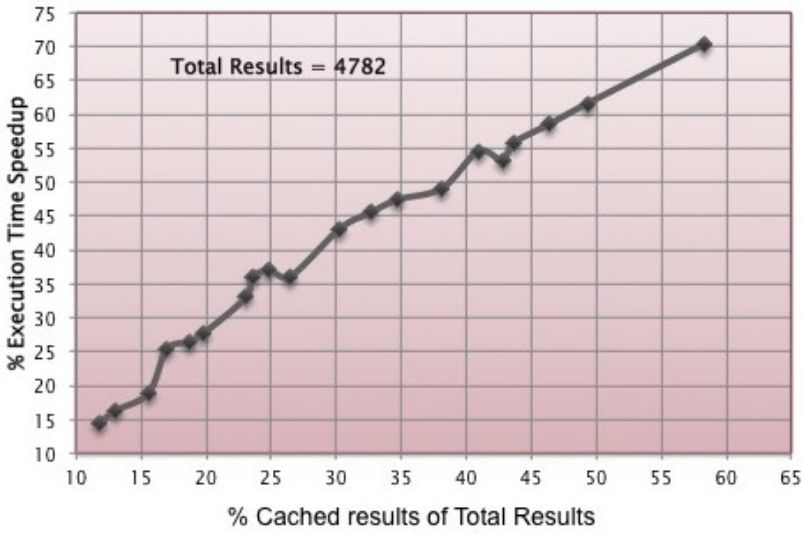


**Figure 7:** How the number of cached articles affects the speedup of a new search

Due the fact that the algorithm stores for each query in the cache a limited set of information relative to the retrieved articles, such as ids, dates and relevance factors, the size of the cache per record in the server memory is kept at minimum. As an example, for caching the 4782 results of the above query, which is a rather generic one with a lot of articles to be found relative, the corresponding row size in the cache database table was measured to be less than 150KB. If we combine the small row size with the periodical deletion of cached query records that expire, the technique can guarantee low storage space requirements in the server.

### 4.5 Expiration Date and Results Accuracy

In the last phase of the experiment, we will examine the impact of selecting a proper expiration time for the cached records on the accuracy and the quality of the final output to the end user. As it was mentioned in the previous paragraph, the proposed algorithm periodically deletes cached records from the corresponding table in the database. The implementation of such an expiration mechanism in the algorithm is essential not only because it helps in keeping the storage space of the server's cache low, but mainly for keeping the accuracy and the quality of the search results at high levels.

Our purpose in this last step of the experiment is to examine how extending the expiration time of the cached records degrades the accuracy of the output result. For this reason, we created a virtual user and constructed a profile for him with favorite thematic categories and keywords. Having no cached data for this user on the first day of the experiment, we had him submit several queries to the system and we cached the results for some of these queries. For the next days of the month, we had the user navigating inside the system by submitting every day several different queries, this time, without caching any of them or expanding the already existing cached results. Among the submitted queries, we included queries identical to the cached ones for comparison to be feasible. The personalization mechanism the portal takes into account the daily behavior

of each registered user (articles he reads, articles he rejects, time spent on each article) and dynamically evolves the profile of the user. For example, it is possible for a user to choose the sports as his favorite category upon his registration, but he may occasionally show an increased interest over science related news. The personalization system then evolves his profile and starts feeding him with scientific news among the sport news and this evolution has an obvious impact in his search sessions inside the system.

In figure 8, we can see how is the accuracy of the search result degraded over the days passing when comparing the actual search results with the cached ones. For our virtual user, on the first day, the average accuracy is obviously 100% as it is the day that the actual queries are cached. Every next day, we get the results (uncached) of the actual queries, which have relevance over 35% to the submitted requests, and we count at average, how many of them existed in the cached queries results. As time passes, the output of the actual queries change (according to the user's evolving profile) and the average percentage of the cached results in the actual output decreases. Until the tenth day of the experiment, we can see that the accuracy is close to 90% to the actual results. However, after the first two weeks the accuracy is degraded at 70% and toward the end of the third week, it is close to 55%. In other words, if the user were to be presented, at this point, with the cached results (cached on the first day), instead of the actual results for his queries, he would see almost, only half of the results that match his changed (since first day) profile. As a conclusion, caching the results of a search for more than two weeks is not a preferable solution for a registered user, as it might significantly produce invalid results, not matching his evolving profile and preferences. However, for unregistered users (guests) of the system, for whom no profile has been formed, an extended expiration date could be used. In our implementation, there is a distinction for registered and unregistered users when checking for cached data, which makes the caching algorithm more flexible.

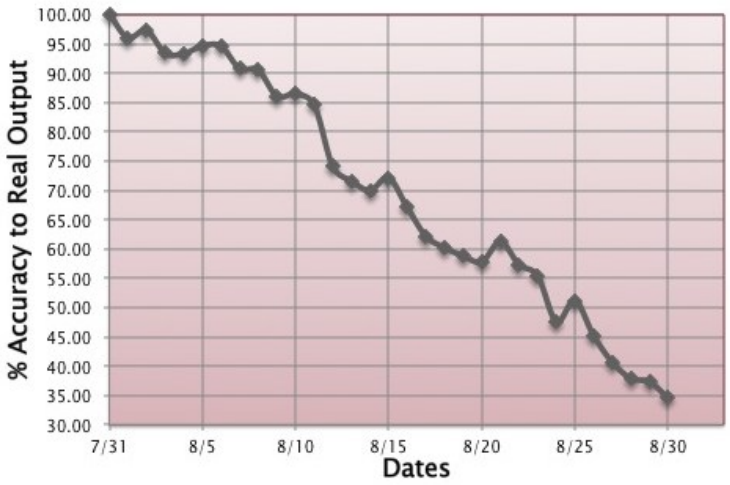


**Figure 8:** How extending date expiration affects the results accuracy

## 5 Conclusion and Future Work

Due to the dynamism of the Web, the content of the web pages change rapidly, especially when discussing about a mechanism that fetches more than 5000 articles on a daily basis and presents them personalized back to the end user. Personalized portals offer the opportunity for focalized results though, it is crucial to create accurate user profiles. Based on the profiles that are created from the peRSSonal mechanism we created a personalized search engine for peRSSonal web portal system in order to enhance the searching procedure of the registered and the non-registered users. Focalizing on the registered users, meaning that we own information about their preferences, we presented a system that is able to personalize the search results on each user's profile and keep the personalization procedure consistent independently of the changes that the profile may undergo. We presented the algorithms and the formulas that lead to personalizing the results and more specifically the ordering of results in order to push the relevant articles to the top of the results for a specific user's profile. Comparing the results to the generic search's results it is obvious that the system is able to enhance the searching procedure and help the users locate the desired results more easily. The system, as every web based system, exports its results on XML format in order to assure a universal output format. For the future what we would like to do is to further enhance the whole system with a more accurate search personalization algorithm in order to make the whole procedure faster and in order to omit any results that are of very low user interest.

## References


Abrams, M., Standridge, C. R., Abdulla, G., Williams, S., & Fox, E. A. (1995). Caching Proxies: Limitations and Potentials. *Proceedings of the 4th International WWW Conference*, (pp. 119-133). Boston.

Bouras, C., Dimitriou, C., Poulopoulos, V., & Tsogas, V. (2006). The importance of the difference in text types to keyword extraction:Evaluating a mechanism. *International Conference on Internet Computing* (pp. 43-49). CSREA Press.

Bouras, C., Kounenis, G., Misedakis, I., & Poulopoulos, V. (2005). A web clipping service information extraction mechanism. *3rd International Conference on Universal Access in Human–Computer Interaction.* Las Vegas: Springer.

Cao, P., & Irani, S. (1997). Cost Aware Proxy Caching Algorithms. *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, (pp. 193-206).

Casaola, E. (1998). *Profusion Personal Assistant: an agent for personalized information filtering on the WWW. Master's Thesis.* Lawrences: The University of Kansas.

Glover, E., Lawrence, S., Brimingham, W., & Andgiles, C. L. (1999). Architecture of a metasearch engine that supports user information needs. *8th International Conference on Information Knowledge Managment*, (pp. 210-216). Kansas City.

Jansen, B. J., Spink, A., & Saracevic, T. (2000). Real life, real users and real needs: A study and analysis of user queries on the Web. *Information Processing and Management* (pp. 207-227). Elsevier.

Jansen, B. J., Spink, A., Bateman, J., & Saracevic, T. (1998). Searchers, the subjects they search and sufficiency: A study of a large sample of EXCITE searches. *In Proceedings of Webnet 98.* Webnet.

Krovetz, R., & Croft, B. W. (1992). Lexical ambiguity and information retrieval. *Information Systems. 10(2)*, pp. 115-141. New York: ACM.

Lawrence, S. (2000). Context in Web Search. *Data Engineering Bulletin. 23*, pp. 25-32. Princeton: IEEE.

Leory, G., Lally, A., & Andchen, H. (2003). The use of dynamic contexts to improve casual internet searching. *ACM Transactions on Information and System Security* (pp. 229-253). ACM.

Lorenzetti, P., Rizzo, L., & Vicisano, L. (1998). *Replacement Policies for a proxy cache*. Retrieved from http://www.iet.unipi.it.

Markatos, E. P. (1996). Main Memory Caching of Web Documents. In *Computer Networks and ISDN Systems* (pp. 893-906).

Oyama, S., Kokubo, T., & Andishida, T. (2004). Domain Specific Web Search with keyword spices. *IEEE Transactions on Knowledge and Data Engineering* (pp. 17-27). IEEE.

Pitkow, J. E., & Recker, M. (1994). A simple, yet robust caching algorithm based on dynamic access patterns. *In Proceedings of the 2nd International WWW Conference.*

Scheuearmann, P., Shim, J., & Vingralek, R. (1997). A Case for Delay-Concious Caching of Web Documents. *6th International World Wide Web Conference.* Roland Wooster & Marc Abrams.

Shen, X., Tan, B., & Andzhai, C. X. (2005). Context Sensitive Information Retrieval using implicit feedback. *28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 43-50). Salvador: ACM.

Shen, X., Tan, B., & Andzhai, C. X. (2005). Implicit user modeling for personalized search. *14th ACM International Conference on Information and Knowledge Managment.* Bremen: ACM.

Silverstein, Henzinger, M., Marais, H., & Moricz, M. (1998). *Analysis of a very large Altavista Query Log.* Technical Report SRC Technical Note.

Spink, A., Jansen, B. J., & Bateman, J. (1998). Users' searching behavior on the EXCITE Web Search Engine. *In Proceedings of Webnet 98.* Webnet.

Teevan, J., Dumais, S., & Andhorvitz, E. (2005). Personalizing Search via Automated Analysis of interests and Activities. *28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 449-456). Salvador: ACM.

Williams, S., Abrams, M., Standridge, C. R., Abdulla, G., & Fox, E. A. (1996). Removal Policies in Network Caches for World Wide Web Documents. *In Proceedings of the ACM SIGCOMM.*

Xu, J., & Croft, W. B. (1996). Query Expansion using local and global document analysis. *19th Annual International SIGIR Conference on Research and Development in Information Retrieval* (pp. 4-11). Zurich: ACM.