# Personal Web Space

Yangjun Chen — Department of Business Computing, University of Winnipeg, Winnipeg, Manitoba, Canada R3B 2E9, ychen2@uwinnipeg.ca

Tony Liu and Paul Sorenson — Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2H1, {tonyliu, sorenson}@cs.ualberta.ca

**Abstract** This paper describes the functional requirement and architecture of a software system called *Personal Web Space* (PWS). A PWS is a system to manage the information from the Web, for either leisure or work related use. Similar to bookmarks, a PWS stores a set of URL addresses but, in addition, it caches on a proxy server pages of high interest. PWS is developed through a evolutionary process that is outlined in the paper. First, a method is prescribed to rank a page (or an URL address) based on the degree of importance to a person as determined by search engine responses and user input. Second, the information stored in a PWS must be refreshed periodically to keep up with the new state of information available on network; therefore a technique is proposed for determining and monitoring freshness. Thirdly, a PWS evolves through enlargement to include more information domains, or refinement to concentrate on smaller sub-domains. The paper describes how enlargement and refinement operations are mapped on to the stored PWS data dictionary. Finally, key-word queries can be issued against a PWS to get interesting information quickly. In this sense, a PWS uses a combination of information caching, information retrieval and bookmarks technique to enhance significantly user performance on the Web.

## 1. Introduction

With the expansion of the Web, more and more comprehensive information repositories can be now visited easily through network. A growing and challenging problem is how to find quickly information of interest to an individual. While navigating through the Web, one may get lost in the maze of hyperlinks [Gr94, Le94] and spend significant amounts of time scanning useless information. A major goal of this research is to investigate mechanisms to help a user to find and then access only the information interesting to him/her. That is, we are interested in supporting high precision queries without suffering significantly from low recall of information. Our proposed method, called PWS, stores a set of URL addresses and some Web pages of high interest, categorized according to subject domains (or topics). It supports bookmarks [bo99], but is more powerful in the sense that

(i) some pages of high interest are cached in a proxy;

(ii) all pages and URL addresses are organized into different domains and each domain is considered as a node in a graph structure used to govern domain/sub-domain relationships, which can then be used to control a PWS evolution process. Also, the navigation within a PWS can be facilitated based on such a domain/sub-domain graph;

(iii) the pages in a PWS are refreshed periodically to keep up with the new state of information from the internet; and

(iv) the key-word queries against a PWS can be conducted just as in an information retrieval system.

Recently, a lot of effort has be directed towards efficient navigation through the Web [Ma99, We99, Kn99, Co99]. In [Ma99], a software tool called *WebGlimps* was presented, which can be considered as a combination of searching and browsing. WebGlimps allows the search to be limited to a "neighborhood" by analyzing automatically the Web pages related to the current one. It provides fast search, efficient indexing and "neighborhood" definition and therefore avoids wasteful use of the internet resources. We adopt the general notion of neighborhood in our PWS approach, but our implementation of neighborhood is quite different. Another software tool, *KnowAll* [Kn99] maintains a huge (100,000 words) lexicon and provides a text-understanding ability in order to return a "right" answer to the question issued. The results are generally superior to those of a normal search engines. A third interesting method, developed in *Copernic* project [Co99], simultaneously consults the best search engines and brings back relevant results with summaries. In addition, duplicate information and dead links can be removed during the searching.

The main goal of the PWS is to provide an optimized view of the internet with respect to relatively specialized domains of personal interest. When compared to the above methods, PWS extends the concept of "neighborhood" by establishing a data dictionary (thesaurus) and performs full-text search against the cached pages using the traditional information retrieval techniques. However, no text-understanding ability is provided in PWS; however, this is an ongoing area of research for PWS.

The rest of the paper is organized as follows. In Section 2, we outline the system architecture of a PWS and provide a brief overview of its functionality. In Section 3, we describe in greater detail the main functionality of a PWS, including the system start-up, the storage refreshment and the PWS evolution. In Section 4, we give a short summary and discuss briefly some future work.

## 2. System architecture

One of our main research direction in PWS is to develop an extendable architecture that supports experimentation with web assistants. As a consequence, PWS is designed and implemented as an extension to ***Sandwich*** [Li99] an O-O framework developed to support personal web assistants. *Sandwich* uses proxy technology to provide basic functions and hooks for plugging in assistants in support of advanced web browsing: in particular, observing and delegating assistants. The relationship between PWS and *Sandwich* is shown in Fig. 1.

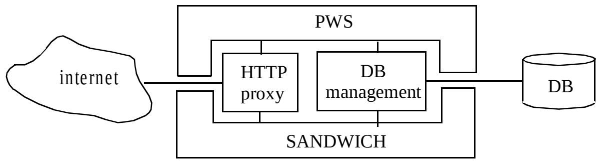


Fig. 1. PWS as an extension to SANDWICH

Among its several functions, *Sandwich* provides support for HTTP stream analysis and an interface to persistence manager (i.e., DB management), both of which are needed in PWS. Using basic *Sandwich* functionality, a proxy is established to handle all HTTP communication between users and the internet. By means of its persistence interface, the data of the system can be stored and manipulated using DB management facilities. As a consequence, the PWS is 'hooked'

[FHLS00] onto *Sandwich*'s HTTP and DB support. To complete PWS, it is necessary to develop its own user interface and some PWS specific modules. More concretely, our PWS has the architecture as shown in Fig. 2.

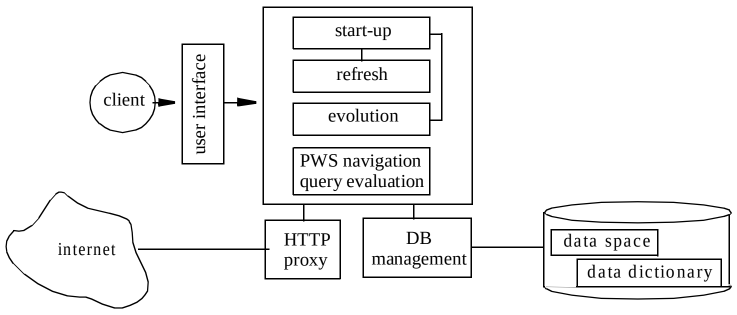


Fig. 2. PWS system architecture

This architecture consists of four parts: interface, PWS core, HTTP proxy and DB-management.

(1) (*interface*) As in most software system, the PWS interface is a suite of application tools which enable users to access the system. In PWS, our interface will support start-up, refreshment and evolution as described below. All information access is conducted using a standard browser (Netscape or Explorer), thereby keeping the management of PWS separate from the access to the PWS.

(2) (*PWS core*) The PWS core consists of four components:

- a start-up-processor to initialize the PWS storage;
- a refresher to change the PWS periodically to keep up with the new state of information on the internet;
- an PWS evolver to govern the interest changes of users (a user's interest may change, expand, or more narrowly focus over time); and
- a mechanism to do PWS navigation and query evaluation.

(3) (*HTTP proxy*) As mentioned above, the HTTP proxy is mainly responsible for communication.

(4) (*DB-management*) The data of a PWS is stored in a database system. As shown in Fig. 1, the data of a PWS consists mainly of two parts: information retrieved from network and a data dictionary which is used to represent the semantic relationships among different information domains. Such data can be easily manipulated using functions provided by a modern database, such as the operations for full text retrieval and maintenance of domain/sub-domain relationships, as well as normal key-word query evaluation.

The third and fourth parts are just a reuse of those implemented in *Sandwich*.

## 3. PWS core

In this section, we describe the main components of a PWS in detail. As outlined in the previous section, a PWS contains mainly four parts which we refer to as the system core: a start-up-processor, a refresher, an evolver and a mechanism for the PWS navigation and the query evaluation. The last part can be implemented using traditional database techniques. Therefore, in the following we focus only on the first three parts.

## 3.1 Start-up

As a first step to build up a PWS, several searching engines such as ‘Yahoo’, ‘Inforseek’, ‘Goolge’, etc. are employed to get information available on network by issuing key words related to a domain which is interesting to a person. To decide which pages are stored in PWS, two tasks have to be conducted: recognizing similar pages and assigning each page a rank to indicate its ‘importance’.

In the following, we first introduce a method to recognize the page similarities in 3.1.1. Then, we discuss the assignment of ranks to pages in 3.1.2.

### 3.1.1 Similarity of pages

Among the pages returned by a searching engine, there may be similar documents obtained from different sites. It is because many documents may be replicated across the web for fast local access, higher availability, or marketing of a site. Therefore, for a PWS, those similar pages should be recognized for reducing space overhead, or for calculating the importance of a page.

One can determine the similarity of two pages in different ways. For instance, one can use the information retrieval notion of textual similarity [Sal83]. One could also use data mining techniques to cluster pages into groups that share meaningful terms (e.g., [PE98]), and then define pairs of pages within a cluster to be similar. A third option is to compute textual overlap by counting the number of chunks of text (e.g., sentences or paragraphs) that pages share in common [SGM95, SGM96, BGM97, BB99, CSG99]. In all schemas, there is a threshold parameter that indicates how close pages have to be considered similar (e.g., according to number of shared words, *n*-dimensional distance, number of overlapping chunks). This parameter needs to be empirically adjusted according to the target application.

All the methods mentioned above don’t, however, pay attention to an important aspect of information: the structure of a page. As we know, a page in HTML or XML format always consists of a hierarchical structure, starting with a root element as shown in Fig. 3(a).

```
<article>
    <articletitle>On the DTD Mapping into OO Schemas</aricletitle>
    <author id=”dawkins”>
                <name>
                     <firstname>Richard</firstname>
                     <lastname>Dawkins</lastname>
                </name>
                <address>
                        <city>Winnipeg</city>
                        <province>Manitoba</province>
                        <zip>R3B-2E9</zip>
                 </address>
     </author>
     <abstract>... database ... SGML ... informatics ... </abstract>
  </article>
```

(a)

```
<article>
    <articletitle>DTD and OO Schemas</aricletitle>
    <author id=”dawkins”>
                <name>
                     <firstname>Richard</firstname>
                     <lastname>Dawkins</lastname>
                 </name>
     </author>
    <abstract>... DB ... XML ... computer science ... </abstract>
  </article>
```

(b)

Fig. 3. Sample XML pages

Such a structure information can be used to speed up page matchings since taking the structure of pages into account the search of similar terms can be limited to small parts of a text. A normaly used technique to explore the similarity of structures is *tree matching*; but it is too strict and a similar page may be filtered out undesirably. So we introduce a more relaxed concept: *tree inclusion*, which can be defined as follows.

**Definition 1** (*labeled tree*) A tree is called a labeled tree if a function *label* from the nodes of the tree to some alphabet is given, or say each node in the tree is labeled. □

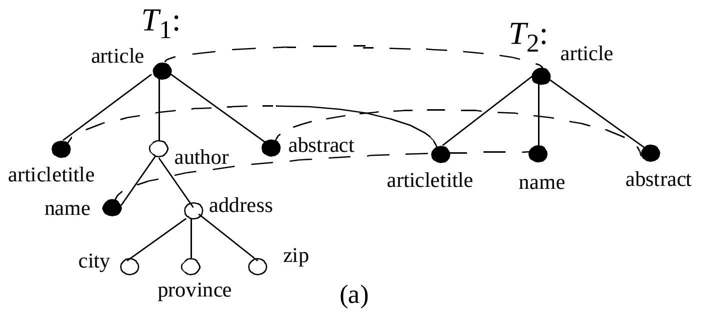
**Definition 2** (*tree inclusion*) Let $T_1$ and $T_2$ be two labeled trees. A mapping $M$ from the nodes of $T_2$ to the nodes of $T_1$ is an embedding of $T_2$ into $T_1$ if it preserves labels and ancestorship. That is, for all nodes $u$ and $v$ of $T_2$, we require that

a) $M(u) = M(v)$ if and only if $u = v$,

b) $label(u) = label(M(u))$, and

c) $u$ is an ancestor of $v$ in $T_2$ if and only if $M(u)$ is an ancestor of $M(v)$ in $T_1$. □

Here, the mapping $M$ can be implemented as a method discussed in [Sal83] or any method used in [SGM95, SGM96, BGM97, BB99, CSG99]. For example, the XML pages shown in Fig. 3(a) and and 3(b) can be represented as two trees shown in the left and right parts of Fig. 4(a), respectively. If a mapping as shown in Fig. 4(b) can be determined, we know that the tree shown in the right part of Fig. 4(a) is included in the tree shown in the left part of Fig. 4(a) (see the dashed lines in the figure). In this case, we claim that the page shown in Fig. 3(b) is similar to the page shown in Fig. 3(a).



$M(T_1.\text{article}) = T_2.\text{article}$
$M(T_1.\text{articletitle}) = T_2.\text{articletitle}$
$M(T_1.\text{name}) = T_2.\text{name}$
$M(T_1.\text{abstract}) = T_2.\text{abstract}$

(b)

Fig. 4. Illustration for tree inclusion

Note that to get an equation such as '$M(T_1$.articletitle) = $T_2$.articletitle', we should check the similarity between 'On the DTD Mapping into OO Schemas' and 'DTD and OO Schemas'. Similarly, to get '$M(T_1$.abstract) = $T_2$.abstract', we have to check the similarity between 'database' and 'DB', 'SGML' and 'XML', and 'informatics' and 'computer science'. Furthermore, we'll utilize the technique discussed in [Sal83] to determine the similarity of $T_1$'s abstract and $T_2$'s abstract based on the common words in these two paragraphs.

The concept of tree inclusion can be used in a variety of ways. For example, we can claim that two pages $P_1$ and $P_2$ are similar if $P_1$ contains a subtree $T_1$ and $P_2$ contains $T_2$ , $T_1 \subseteq T_2$ (or $T_1 \supseteq T_2$), and at the same time both the ratios: $|T_1|/|P_1|$ and $|T_2|/|P_2|$ are larger than a certain shreshold, where $|T_i|$ and $|P_i|$ ($i$ = 1, 2) represent the numbers of nodes in $T_i$ and $P_i$, respectively. It is one of our forthcoming researches to explore different ways to use the concept of tree conclusion to recognize effectively the similarity of pages.

### 3.1.2 Page storage for start-up

The appearance number *f* of a page in the result set produced for different searching engines can be calculated. We are experimenting with several approaches to calculating *f* based on how high a page appears in the result list of each of the search engines. In addition to a computed *f* value, To each obtained page, a user can attach a user interest value *v* to selected pages. Typically, during start-up all pages are assigned *v* value of 0. As PWS matures, certain pages will be either marked explicitly by the users as important using a bookmark (in which case *v* is set to an integer greater than zero) or will be implicitly assigned a *v* value based on how often the user accesses that page over a given period of time. Expressed in terms of *v* and *f*, we assign each page a value *k* to represent the degree of interest as follows. We sort the web pages in terms of $t = v + f$ such that a page $b_1$ appears before another page $b_2$ if $t_1 \geq t_2$, where $t_1$ and $t_2$ represent *t* values associated with $b_1$ and $b_2$, respectively. Then, each page *b* will get an order number *k* according to the sorted page sequence. The pages with the same value of *t* will have the same *k* value. Hence, a page with a *k* value of *i* can be considered as an *i*th most interesting page to that person. Using Zipf's law [Zi29, BCF99] as an estimator of the relative probability *p* that a person accesses a page with *k* value as follows:

$$p = 1/k.$$

(Note that Zipf's law has often been used as an estimator of the expected frequency of page accesses in a virtual memory paging environment [BCF99].)

Furthermore, we want to rank a page based on more aspects that can affect the degree of importance to a person. Let *cf* ($0 \leq cf \leq 1$) be the rate of modification of a page and *sz* the page size. The rate of modification is the number of changes of a page over a given period of time. Some web pages provide such information in their page. If we cannot get such information, we set *cf* to 1 if the corresponding page is dynamic (e.g., a page constructed on the fly); otherwise, *cf* is set to 0. The page size *sz* is counted in Kbytes. We compute the rank of this page *h* ($0 \leq h \leq 1$) as follows:

$$h = (1 - cf) \cdot \frac{p}{max\{\log sz, 1\}}$$

The meaning of *h* is that a page with a higher value of *h* will be more likely to be stored, which reflects the intuition that a page frequently changing (*cf* approaches 1) and with a large size should not be cached. In the above formula, we use the logarithm of *sz* so that *h* values are not dominated by the page sizes. In contrast, a small page that rarely changes and is of high interest should be kept locally. Note that if a page is dynamic it should not be cached and should therefore be assigned a *cf* of 1. We use *Sandwich* HTTP analyzer to detect dynamic pages. The size of a page *S* is represented by a parameter *sz* which is computed as follows. If $|S|$ is, for example, equal to $k \times 100$, where k = 1024 bytes, then $sz = 100$.

Expressed in terms introduced above, a start-up process can be done as follows.

**Procedure** *start-up*($kw$; $en_1, ..., en_k$)

/**kw* represents a set of key words; $en_1, ..., en_k$ are the search engines used*/

1. Use $en_1,..., en_k$ to retrieve information by issuing *kw* related to a certain domain of interest.
2. Let $S_j$ be the page set obtained using $en_j$. Let $S = S_1 \cup ... \cup S_k$. (Note that *S* is a multi-set; *i.e.*, a member of it may appear several times.)

3. Assign each different $b \in S$ a value $v$ and compute its appearance number $f$ (i.e., the number of times $b$ appearing in $S$). After that, $k$ and $p$ can be calculated.
4. Compute $h$ for each different $b \in S$. For the start-up, we take $cf = 0$ for each different $b$ unless $b$ is dynamic, in which case $cf = 1$.
5. In terms of $h$ values, store $S$ as shown in Fig. 5(a).

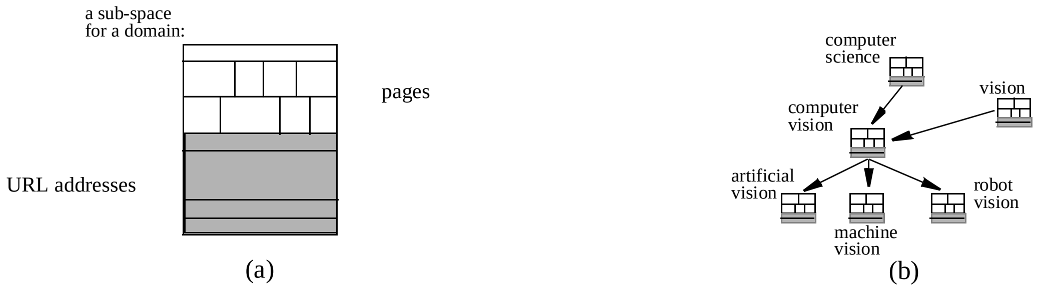


Fig. 5. PWS storage

The PWS space for a domain is divided into two parts. The first (lower) part is all the URL addresses of $b$'s ($b \in S$). The second (upper) part is a set of pages with higher $h$ values.

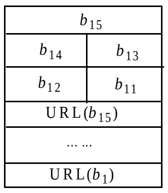
We note that at the very beginning, a user is not able to know the change frequency of a page. As a consequence, we just assume that each obtained page has the same values of $cf$ (= 0). But during the course of operations, a user may observe the modification rate of some pages. Then, he/she can assign the corresponding values to them based on those observations.

A PWS may contain information on several domains. The information on one domain constitutes a sub-space. Then, we construct a directed graph (called a *domain graph*) in which each node represents a sub-space and each edge represents a domain/sub-domain relation as shown in Fig. 5(b). (Such a graph can be constructed using the data dictionary if the relevant information is available in it; or constructed with interference of human beings.)

**Example 1.** Assume that $S = \{s_1, ..., s_{10}\}$ and $T = \{t_1, ..., t_{10}\}$ are two sets of pages obtained respectively using two search engines $en_s$ and $en_t$ by issuing the same key word set search. Assume that $s_i = t_i$ for $1 \le i \le 5$. Then $S \cup T = \{s_1, ..., s_{10}, t_6, ..., t_{10}\} = \{b_1, ..., b_{15}\}$, where $b_i = s_i$ for $1 \le i \le 10$ and $b_j = t_j$ for $10 < j \le 15$. Then, the appearance number of $b_i$ is $f(b_i) = 2$ for $1 \le i \le 5$ and $f(b_j) = 1$ for $6 \le i \le 15$. Assume that we assign each $b_i$ a value $v(b_i) = i$ for $1 \le i \le 15$. This is quite unpractical, but it demonstrates the computation process. The values of $k$ can be calculated as shown in Fig. 5(a). In terms of $k$ values, $p$ values can then be calculated. Assume that for each $b_i$ its $cf$ value is 0 and its size $|b_i|$ is $(16 - i) \times$ Kbytes (then its $sz$ is $i$). We give the values of $h$ for each $b_i$ ($1 \le i \le 15$) in Fig. 6(a).

|  | $f$ | $v$ | $t = v + f$ | $k$ | $p$ | $cf$ | $sz$ | $h$ |
|---|---|---|---|---|---|---|---|---|
| $b_{15}$ | 1 | 15 | 16 | 1 | 1.000 | 0 | 1 | 1.0000 |
| $b_{14}$ | 1 | 14 | 15 | 2 | 0.500 | 0 | 2 | 0.5000 |
| $b_{13}$ | 1 | 13 | 14 | 3 | 0.333 | 0 | 3 | 0.3050 |
| $b_{12}$ | 1 | 12 | 13 | 4 | 0.250 | 0 | 4 | 0.1800 |
| $b_{11}$ | 1 | 11 | 12 | 5 | 0.200 | 0 | 5 | 0.1250 |
| $b_{10}$ | 1 | 10 | 11 | 6 | 0.166 | 0 | 6 | 0.0920 |
| $b_9$ | 1 | 9 | 10 | 7 | 0.142 | 0 | 7 | 0.0730 |
| $b_8$ | 1 | 8 | 9 | 8 | 0.125 | 0 | 8 | 0.0603 |
| $b_7$ | 1 | 7 | 8 | 9 | 0.111 | 0 | 9 | 0.0505 |
| $b_6$ | 1 | 6 | 7 | 10 | 0.100 | 0 | 10 | 0.0434 |
| $b_5$ | 2 | 5 | 7 | 10 | 0.100 | 0 | 11 | 0.0418 |
| $b_4$ | 2 | 4 | 6 | 11 | 0.090 | 0 | 12 | 0.0362 |
| $b_3$ | 2 | 3 | 5 | 12 | 0.083 | 0 | 13 | 0.0320 |
| $b_2$ | 2 | 1 | 4 | 13 | 0.077 | 0 | 14 | 0.0290 |
| $b_1$ | 2 | 1 | 3 | 14 | 0.072 | 0 | 15 | 0.0260 |

(a)



(b)

Fig. 6. Exemplary storage

If there is room to cache only five pages after all URL addresses are put into the PWS space, the map of the storage space will be as shown in Fig. 6(b).

## 3.2 Refresh

A PWS should ideally provide access to the latest information. To this end, a periodical refreshment of the storage is needed. Two different refreshment operations may be performed:

- refreshment of cached pages
- replacement of URL addresses

Nowadays a page on Web may have the following parameters: *max-age*, *min-age* and *expiring-time* for the management purpose, where *max-age* is the longest time the page can exist, *min-age* is the shortest time the page can exist and *expiring-time* indicates when the page becomes invalid. A user can also assign each cached page a *time-tag* to indicate its replacement period. When a page reaches its replacement period, it should be refreshed. Obviously, such parameters can be used to implement the operation to refresh cached pages. The following is a simple algorithm to do this task, which is in fact a modified version of Squid algorithm discussed in [Sq99]. In the description of the algorithm, by the age of a page, we mean the time that page exists in PWS.

```
Procedure marking(pa)
input: a cached Web page pa;
output: pa will be marked with “fresh” or “stale”;
    begin
        if max-age exists for pa then
            if pa’s age is more than max-age then {mark pa with “stale”; return;}
        if min-age exists for pa then
            if pa’s age is less than min-age then {mark pa with “fresh”; return;}
        if expiring-time exists for pa then
            if pa has already expiredthen {mark pa with “stale”; return;}
        if pa reaches the replacement period (time-tag) then {mark pa with “stale”; return;}
        mark pa with “fresh”
    end
```

By the above algorithm, a cached page will be marked with “fresh” or “stale”. In terms of these marks, a refreshment operation can be performed as follows.

**Procedure** *refreshment*(*P*)

```
input: a set of cached Web pages;
output: a new set of Web pages to be cached;
    begin
        for each $pa \in P$ do
          {if pa is marked with "stale"
           then remove pa and get the new version of pa using the corresponding URL address;}
    end
```
For the replacement of URL addresses, we simply call the start-up-processor, but without caching any concrete page.

## 3.3 PWS evolution

The PWS evolution is another important function of PWS. As time goes on, a user's interest may change from a domain to another. For this purpose, the system provides a built-in data dictionary to govern the relationship among different domains and thus control the domain transfer. In addition, a user is able to insert entries into or delete entries from a data dictionary. Generally, a data dictionary is a set of tree structures and each node in a tree is a pair of the form: $<\alpha, \beta>$, where $\alpha$ represents a domain name and $\beta$ is a list of key words related to the domain. There is a path from node *a* to node *b* in a tree structure if *b* represents a sub-domain of *a*. In Fig. 7, two tree structures in a data dictionary are given.

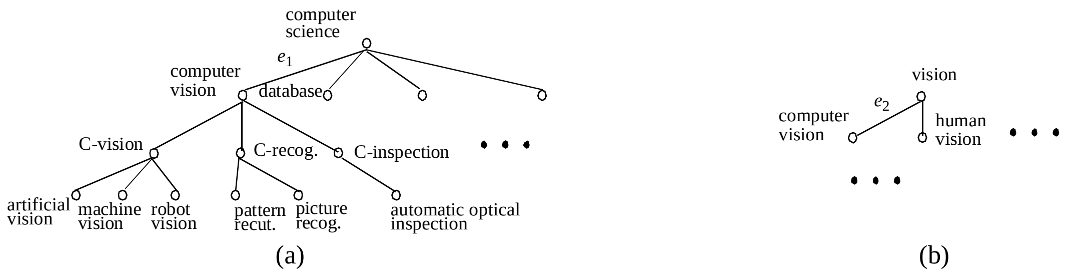


Fig. 7. An entry in a data dictionary

*- PWS enlargement*

The PWS enlargement is used to govern the process of the domain changing from a smaller one to a bigger one. One the one hand, the domain enlargement is an important character of interest transfer. On the other hand, the relationship of domain/sub-domain is a simple principle of organizing PWS spaces. Thus, it should be supported.

The PWS enlargement can be made along a path in a tree structure of the data dictionary or using the key words given by a user. In the former case, the key words associated with the parent (or ancestor) node of a node whose corresponding domain will be enlarged, will be used to search the Web. We further distinguish between two enlargements:

- single enlargement and
- multiple enlargement

If a concept *a* (representing a domain) appears only in one tree structure in the data dictionary, the enlargement for *a* must be a single enlargement. If it appears in several tree structures simultaneously, a multiple enlargement will be performed. That is, along the paths in the different tree structures, several enlargements will be conducted.

Formally, a single enlargement can be described as follows. Let $T$ be a tree structure (an entry in the data dictionary), $a_l$ be a node in $T$ and $p = a_1 \rightarrow ... \rightarrow a_l$ be a path in $T$. (Note that $a_1$ is not always the root of $T$. It is determined by the user.) Assume that the concept associated with $a_l$ appears only in $T$ and $S$ is the node in the domain graph, representing the sub-space for $a_l$. Then, a single enlargement can be performed by doing the following two operations:

$S' \leftarrow$ start-up($kw$; $en_1$, ..., $en_k$);

establish an edge $e = (S', S)$ in the domain graph; mark $e$ with $p$;

where $kw$ represents the key word list associated with $a_1$, $en_1$, ..., $en_k$ represent $k$ search engines used and $S'$ represents the PWS sub-spaces for $a_1$. The relationship between $S$ and $S'$ is represented as an edge in the domain graph.

Similarly, a multiple enlargement can be described as follows. Let $T_1$, ..., $T_m$ be $m$ tree structures in the data dictionary containing a same concept. Assume that its corresponding nodes in $T_1$, ..., $T_m$ are $c_{11}$, ..., $c_{l1}$, respectively. Let $p_i = c_{ij_i} \rightarrow ... \rightarrow c_{i1}$ be a path in $T_i$ ($1 \le i \le m$). We do the following operations to implement a multiple enlargement:

$S \leftarrow$ start-up($kw_1 \cup ... \cup kw_m$; $en_1$, ..., $en_k$);

establish an edge $e = (S', S)$ in the domain graph; mark $e$ with $\{p_1, ..., p_m\}$;

where $kw_1$, ..., $kw_m$ represent $m$ lists of key words associated with $c_{1j_1}$, ..., and $c_{lj_m}$, respectively.

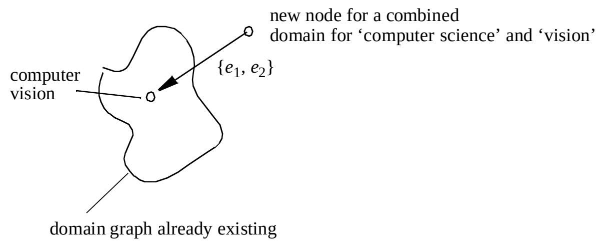
We illustrate the above operation using the following example.

**Example 2.** Consider the two tree structures shown in Fig. 7. Assume that $kw_1$ and $kw_2$ are two lists of key words associated with 'computer science' and 'vision', respectively. The multiple enlargement of domain 'computer vision' using search engines 'Yahoo' and 'Inforseek' can be done as shown in Fig. 8(a). The result is a sub-space for a new domain comprising 'computer science' and 'vision' as shown in Fig. 8(b).

$S \leftarrow$ start-up($kw_1 \cup .kw_2$; Yahoo, Inforseek);

establish an edge $e = (S, S')$; mark $e$ with $\{e_1, e_2\}$;

(a)


new node for a combined
domain for 'computer science' and 'vision'
{e1, e2}
computer vision
domain graph already existing

(b)

Fig. 8. Illustration for multiple enlargement

By this multiple enlargement, the domain 'computer vision' is enlarged along $e_1$ in the tree shown in Fig. 8(a) and along $e_2$ in the tree shown in Fig. 8(b). The result of this enlargement is a combined domain for 'computer science' and 'vision'. To represent the relationship between this new domain and the domain 'computer vision', a new node $v$ for it and a new edge $e$ (labeled with $\{e_1, e_2\}$) connecting $v$ and the node for 'computer vision' will be constructed. (See Fig. 8(b) for illustration.)

Alternatively, a user can choose a list of key words by himself. Corresponding to this list of key words, a new domain is created by the user and therefore should be named, which is then inserted into the domain graph which is used to govern the domain/sub-domain relationships.

- *PWS refinement*

The PWS refinement can be performed in a similar way to the enlargement, but reversed. This operation is needed when the user narrows his/her interests.

A refinement is always done down a path in a tree structure in the data dictionary. Let $T$ be a tree structure, $a_1$ be a node in $T$ and $p = a_1 \rightarrow ... \rightarrow a_l$ be a path in $T$. Then, a refinement from $a_1$ to $a_k$ can be performed as follows:

```
$S' \leftarrow$ start-up($kw$; $en_1, ..., en_k$);
establish an edge $e = (S, S')$; mark $e$ with $p$;
```

where $kw$ represents the key word list associated with $a_l$. This is just a reverse process of the single enlargement.

## 4. Summary and future work

In this paper, the architecture and functionality of a software system: PWS is discussed in detail. Like bookmarks, a PWS contains a set of URL addresses of interest but with some concrete Web pages cached. In addition, a lot of functions are supported in a PWS to facilitate the access, the manipulation and the management of interesting information from the internet. A PWS can be started up by retrieving information of interest on network, which will be organized into a domain/sub-domain graph stored locally. The storage of a PWS can also be refreshed periodically to keep up with the new state of the informationon Web. Further, a PWS can be evolved to reflect the interest change of users. At last, a PWS can be enquired by issuing queries just as in an information retrieval system. In this way, a PWS works as a combination of information caching, information retrieval and bookmarks.

As a future research, more aspects affecting the storage strategy of PWS should be explored. For example, the reliability of pages should be taken into account for page storing. It is intuitive that a page with lower reliability (*e.g.*, the source server is often down or unavailable) may not be worthwhile as an information source for a PWS. This observation can be integrated into the formula to compute $h$ values as follows. Let $r$ ($0 \leq r \leq 1$) be a value to represent the reliability of a page. We reconstruct the way to calculate $h$ as shown in the formula below:

$$h = (1 - cf) \cdot \frac{p \oplus r}{max\{\log sz, 1\}},$$

where $p \oplus r = p + r - p \cdot r$. (Note that, if $0 \leq p \leq 1$ and $0 \leq r \leq 1$, we have always $0 \leq p \oplus r \leq 1$.) From this we can see that a page of higher reliability will has a higher $h$ value. The problem is how to determine $r$ values, i.e., how to check automatically whether an information resource is reliable or not. For this purpose, we need a mechanism to observe the behavior of an information resource and then “reason” how high it is reliable. Another interesting parameter is “interest to public”, which can be obtained by checking how many times a page is linked by others, which needs another mechanism to trace the Web.

In addition, we are planning to enhance the full-text searching ability of the system with text-understanding, which will make a PWS more intelligent and therefore the cached pages will fit the user's needs more exactly.

## References

BB99 K. Bharat and A.Z. Broder, Mirror, Mirror, on the web: A study of host pairs with replicated content, in *Proc. of 8th Int. Conf. on World Wide Web (WWW'99)*, May 1999.

BCF99 L. Breslau, P. Cao, L. Fan, G. Phillips and S. Shenker: Web Caching and Zipf-like Distributions: Evidence and Implications, http://www.cs.wisc.edu/~cao/papers/zipf-implications.html.

BGM97 A.Z. Broder, S.C. Glassman and M.S. Manasse, Syntactic clustering of the web, in *Proc. of 6th Int. World Wide Web Conference*, April 1997, pp. 391-404.

bo99 bookmarks home page: http://www.whatis.com/bookmark.htm.

Co99 Copernic: http://www.copernic.com.

CSG99 J. Cho, N. Shivakumar, H. Garcia-Molina, "Finding replicated web collections," http://dbpubs.stafford.edu/pub/1999-64.

FHLS00 Froehlich, G., Hoover, H.J., Liu L. and SORENSON, P.G.: "Reusing Application Frameworks Through Hooks," to appear in *Object-Oriented Application Frameworks*, M. Fayad, R. Johnson editors. John Wiley & Sons, New York, NY.

Gr94 I.S. Graham: HTML-documentation and style guide, http://www.utirc.utoronto.ca/HTMLdocs/NewHTML/htmlindex.html, 1994.

Kn99 KnowAll: http://www.worldfree.net.

Le94 T.B. Lee: RFC 1738: Uniform Resource Locators, http://www.w3.org/hypertext/WWW/Addressing/rfc1738.txt, Dec. 1994.

Li99 W. Liew: Sandwich: A Personal Web Assistant Framework, master thesis, Department of Computing Science, University of Alberta, 1999.

Ma99 U. Manber: WebGlimps - Combing Browsing and Searching, Department of Computer Science, University of Arizona, http://ftp.cs.arizona.deu/people/udi/webglimps.ps.Z.

PE98 M. Perkowitz and O. Etzioni, Adaptive web sites: automatically synthesizing web pages, in *proc. of 15th National Conf. on Computer and Human Interaction (CHI'97)*, 1997.

Sall83 G. Salton, *Introduction to modern information retrieval*, McGraw-Hill, New York, 1983.

SGM95 N. Shivalumar and H. Garcia-Molina, SCAM: a copy detection mechanism for digital documents, in *proc. of 2nd Int. Conf. on Theory and Practice of Digital Libraries (DL'95)*, Austin, Texas, June 1995.

SGM96 N. Shivalumar and H. Garcia-Molina, BUilding a scalable and accurate copy detection mechanism, in *proc. of 1st Int. Conf. on Digital Libraries (DL'96)*, Bethesda, Maryland, March 1996.

Sq99 Squid: http://www.squid-cache.org.

We99 WebGlimps: http://webglimps.org.

Zi29 G.K. Zipf: *Relative frequency as a determinant of phonetic change*, Reprinted from the Harvard Studies in Classical Philiology, Vol. XL, 1929.