# Personal Information Systems

Gaurav Veda, Sarvesh Dwivedi
Email: {gveda, dwivedi}@cse.iitk.ac.in
Guide: Prof. Sumit Ganguly
Guide's Email: sganguly@cse.iitk.ac.in

Department of Computer Science & Engineering
Indian Institute of Technology
Kanpur, UP, INDIA - 208016

***Abstract*— We are living in the information age. Infact, we are now facing the problem of information overload. There is so much information present in this world, that we often find ourselves trying to find a needle in a haystack. On the web, we often succeed in retrieving what we want by using search engines like Google that make use of its underlying graph structure. However, the problem remains largely open for desktop systems, inspite of a few recent softwares.**

**Here, we present an information model which addresses this problem. Using our model, users can retrieve even those documents which do not contain the search keywords, but contain information that is *related* to the search keywords. Since our model is personalized, the property of two documents being related depends on the user of the system. We also introduce a notion of activities which, we believe, is the way in which humans think about information. Due to the little computation involved, our model enables us to answer queries efficiently and requires very little storage space.**

## I. Introduction

Today, we are witnessing an information revolution. We are being constantly flooded with information, be it on the internet or due to other sources (like emails), and it is becoming increasingly difficult to manage such huge amounts of information. An even more formidable task is to find something relevant through this great mass of data. The problem has more or less been addressed for the internet, with the advent of various good search engines (like Google) which can find out the required data given a few keywords. Most of these search engines make use of the well understood hyperlink structure of the web.

This problem of efficient information retrieval however, persists for the data stored on the hard disks of our computers. One of the primary reasons for this is the absence, till date, of any information model that relates documents for a desktop system in the way that hyperlinks relate documents on the web. Moreover, the problem is escalating day by day with the burgeoning size of secondary storage. Consequently, it is becoming increasingly difficult for a user to store data logically and access it easily.

We believe that each of the units of information on a user's hard-drive collectively form an information system that is much more than a decoupled collection of information units. The information system that is built out of these units is usually not modeled in explicit form at any place, but rather, it resides in the mind of the user. Currently, a user stores her information in a hierarchical fashion using files and directories and tries to encode this information system in their naming conventions. However, the problem with this approach is that, firstly it is very difficult to maintain and is not scalable. Secondly, what should a user do when a given piece of data qualifies for being kept in multiple directories, and thirdly, what if the user wishes to arrange the data according to a completely new classification scheme or someone else takes over the system (eg. in a corporate setting). The aim of this project is to solve this problem, so that the user does not have to care much about how to arrange data and is still able to easily retrieve the required data. To do this, we propose a new information model for storing information in a desktop system.

## II. Motivation

The aim of this project is to build a system which does automatic classification and then allows a user to retrieve a document even if she does not know the exact words it contains, but only has an idea what the document is about. In other words, we want to build a system which is able to capture the way in which humans think about information. We feel that the restrictions imposed by the hierarchical directory system are very artificial. In a hierarchical directory structure, a file is supposed to placed in only a single directory. However, this doesn't make much sense, since human beings tend to classify a single information unit into various categories. For example, a user will associate a paper on Peer to Peer Systems with both Networks and Databases. Currently, there is also no mechanism to capture the temporal relation between data items. We believe that people often recall a particular piece of information or an event, using events that were done by them at the same time, even though in terms of content the events might be totally unrelated. As an example, a person might relate a book on Algebra to Switzerland, because she read it while on vacation there. Thus there are various types of associations. This cannot be captured in the directory structure. Also the directory structure scales badly because the user has to manually create directories and links to files (if she wants a document to appear at more than place). Arranging and maintaining data on a 80GB/160GB (the current size of

hard-disk on a recently bought personal computer) requires too much time and energy which users are rarely able to devote.

We want to build a system in which the user doesn't have to change directories. Infact, she doesn't even need to know how are files stored on the computer internally. Whenever she wants to retrieve a file, she just queries for it using some words (not necessarily keywords) and the system displays a ranked list of matching documents. To save a file, she just has to specify its name and optionally some words that describe it. The system then classifies it and stores it in a way that makes subsequent retrieval of the file easy.

As stated above, we want to enable non-keyword based search. All the systems that are in place today, rely on a more or less purely keyword based scheme to search for information. Such a system might very well cater to the kind of queries that we make on the web. This is because on the web, we are mainly interested in finding information that has *not been entered by us*. However, even on the web, we are sometimes plagued by the problem of being unable to retrieve the information we seek, mainly because of purely keyword based search systems. This problem takes gigantic proportions when it comes to a desktop. On a desktop, we often want to make queries which simply cannot be answered, if we just do a keyword based lookup. Using the words inputted by the user to describe a document, we can say that a given document is related to another one. Using all this information, the system will come to have an idea of how the user relates various things, thus paving way for a personalized system.

From our own experience and observing other users, we believe that an average user looks atmost at two-three pages of search results. This translates to approximately 20-30 search results. This implies that a good ranking algorithm is the heart of any search engine. Without a good ranking scheme, a search engine is useless, howsoever fast it might be. Our aim is to calculate the most relevant results as quickly and efficiently as possible and present them to the user.

Once the whole system is in place, it would simplify information retrieval and make it more effective. For example, a professor would be able to retrieve a question that he wrote for a database course exam some time back if he just remembers the basic idea behind the question. She will not have to recall where did she store it. She would just have to type in a few words describing the question (some or all of which might not even be contained in the document), and the system will come up with the required document.

## III. Related Work

As of now, there are desktop search tools available from a number of vendors viz. Google[1], Yahoo[2], Apple[3], Copernic[4] and others. All of these systems essentially do a full content indexing of the data present on the system and after this is done, the user can do a keyword based search on the index for whatever data she wishes to find. Although this forms the basis of all these systems, they also provide a few additional functionality.

Along with doing complete content indexing, these systems also integrate mail clients, chat clients and web browsers, so that users can search their mails, chat logs and browsing history along with other files. Another feature of the above mentioned systems is that they have format specific readers for popular file formats ilke MS Word, MS Excel etc. Using these format readers, they are able to extract metadata from the documents and index this metadata also. This enhances the quality of search. Additionally, during search this feature can be used to find specific types of files. Some of the tools provide another feature which is the "up-to-the-moment accurate" search capability. This capability means that within a short time after the creation/addition of a new file to the system or the modification of an existing file, the tool will be able to index it and users will be able to search through the new/modified data.

The tool Spotlight, which is offered by Apple in their upcoming operating system Tiger, has a special feature of saving search results in "Smart Folders". Using this feature, a user can arrange her data in a logical way and surpass the artificial barriers of directory structure as the user can now view documents in a personalized way which is also dynamic. Also, these smart folders get updated as new data arrives on the system or old data gets deleted. This takes care of classification to some extent, though in no way this classification is automatic.

Another tool is Haystack[5], which is an information management tool. It lets users view their data in whatever way they wish using a semi-structured data model. This data model comprises of actual documents as well as the metadata of these documents. The user can build various relations among documents. This is done using Resource Description Framework (RDF)[6]. Haystack also uses format specific readers, thus enabling the user to view a document as only an information unit. It provides support for annotating documents. Additionally, a user has the flexibility to define new data types which can have personalized attributes.

The problem with the search tools discussed above is that they do only keyword-based search and as stated earlier, this does not suffice for a desktop system. Also, except Spotlight, they lack the capability for any classification as such. On the other hand, though Haystack lets users arrange and view data in a personalized fashion, it demands too much from the user if she wishes to attain any reasonable classification, since there is no automatic classification done as part of the system.

One of the problems with plain keyword based search, is the estimation of relevance of a keyword to a document. One obvious way can be to use the number of times a word occurs in the document. However, the number of occurrences is not a true indicator of the relevance of the word to the actual content of the document. Moreover a given keyword might not occur in the document, even though it is highly relevant to the content.

For example, suppose that there is a document containing a discussion on the primal-dual algorithm for linear programming, and further suppose that the keyword(s) simplex algo-

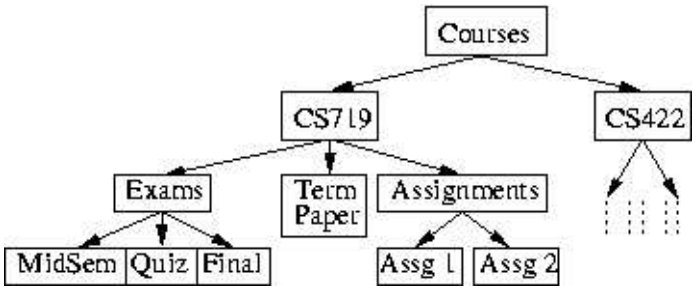


Fig. 1. An example of the activity hierarchy

rithm does not appear in the document. However, documents that are highly relevant to simplex algorithm are relevant to the current document, since, the simplex algorithm is highly relevant to linear programming. The solution in the context of the web, as devised by Google (and its variants), is to model documents as web pages, and to use hyperlinks from one page to another to recursively quantify relevance. So, continuing the example above, although a web page might not contain the keyword(s) 'simplex algorithm', there might be many links to this page from other pages where this keyword occurs in the hyperlink text. However, there is no direct notion of hyperlinks in a personal information system, and hence, a Google like search and retrieval mechanism does not extend obviously to this domain.

In our model, we also do a full content indexing of the documents in the system. Additionally, by using format specific readers, various formats can be supported by us. Document classification in our system is done automatically with optional user intervention. The user can decide her level of involvement in the classification process. Therefore, we support all the features provided by the existing search utilities. In addition, we also provide the capability of non keyword based search and the concept of activities (as is described later). This makes our system radically different from any existing search system.

## IV. Our Information Model

We conjecture that human beings tend to think and organize information in terms of activities and information classes.

We teach classes, enroll in courses, take vacations, participate in projects etc. Each of these actions constitute an activity. Activities have a clear beginning and end in terms of time eg. the course CS625 (being offered this semester) started in January 2005 and will end in May 2005.

At a given point of time, we indulge in a number of activities. Corporate activity is often organized into a set of concurrently running projects with reasonably well-defined goals.

New activities are often related to older activities eg. if an instructor teaches the course CS719 every fall semester, then the activity of teaching the course in Fall 2004 is related to the activity of teaching the course in Fall 2005.

Information processed during the span of an activity is typically related. The challenge lies in forming a model that is able to capture this relation in an automated system.

An information class is like an activity, but it does not have any start time or end time associated with it. They are timeless things. eg. book writing is an activity, but a book belongs to an information class.

An information unit is primarily classified based on its *content*. Whenever a new file enters the system, it is first classified into one or more activity or information class.

Each activity or information class has an associated description with it. This description consists of a set of keywords along with a number between 0 and 1 (which we call 'relevance') associated with each keyword. eg. let us consider the course 'CS719' (An introduction to Data Streaming) to be an activity. Its description tag might look like:
{data streams (1.0), algorithms (0.7), randomized algorithms (0.9), database (0.3)}

As stated earlier, the relevance of a keyword is a number between 0 and 1. It is a measure of how well a keyword describes an activity or how relevant is the keyword to that particular activity. In the above example, we observe that the course CS719 is accurately described by the keyword 'data streams'. The course comprises mainly of randomized algorithms and hence that keyword has a relevance of 0.9. The set of keywords together with their relevance scores, and the start and end time, forms the *signature* of an activity. An information class is also associated with a similar signature, that is, keyword, relevance score pairs. In this sense, an activity is distinguished from an information class by the presence of the time field, that is, duration. We discuss the relation between activities and information classes in greater detail in the section 6 below.

### A. Information Classes Hierarchy

Information classes may have a containment hierarchy which resembles a directed acyclic graph. For example, a book consists of a foreword, a preface, a table of contents, chapters, bibliography and an index. A chapter is made up of sections, sections have the textual parts and figures, and so on. The above description can often be adequately captured using an XML schema definition. We therefore propose that non-atomic information unit classes, that are built out of instances of simpler information unit classes, are specified using an XML schema definition (or, equivalently, using an object-oriented

schema definition).

### B. Activity Hierarchy

Akin to information classes, activities too can be organized into an activity - sub-activity hierarchy. For example, the course *CS719* activity may have *Problem Set* as a sub-activity; analogously, *Mid Semester Exam 1, Mid Semester Exam 2, End Semester Exam* and *Course Projects* may be sub-activities of the activity *CS719*. The sub-activity *Problem Set* may have individual problem sets, say *Assignment 1* through *Assignment 5* as further sub-activities. Each sub-activity is an activity, and therefore, is specified by a set of keywords with weights and a duration.

It appears to be reasonable to label a sub-activity with a subset of the set of keywords that its parent activity is labelled with, possibly with changed relevance measures. In addition, it may be labelled with a few other keywords. For example, the Problem Set sub-activity may have *problem set* and *assignment* as key words.

**Schema definition for activities:** Some activities may have an XML schema associated with them. For example, the activity *Course* (of which *CS719* is an instance), has simple attributes, such as name of the course, instructor name(s), list of student ids enrolled in the course, their e-mail ids etc. The sub-activity *Problem Set* has Date of handout and due date, respectively as attributes. This means that each instance of the sub-activity *Problem Set* has these attributes (unless they are null). In general, the information wizard provided by the system allows the specification of new classes of activities. In addition, the system may be provided with a generic set of information classes that are useful to a wide variety of users, and specialized information classes for use by specific sets of users. For example, doctors, lawyers, academician, software developers, accountants etc. will possibly have a different view of the world, that is encapsulated by a set of domain-specific information classes. The usefulness of a personal information system is expected to be greatly enhanced by having a library of domain-specific information classes and activities.

## V. Problem Definition

Our system is based on the information model described by us in Section 4. We need to address a host of problems so that our system is able to do automatic document classification, non-keyword based retrieval of documents and is able to support activities. This invovles, firstly, classification of a new/modified document so that it can enter our information model ie. it has to be put it in its proper place(s) in the activity model. Later, while searching, we need to rank the document. We use relevance propagation as the ranking methodology, using which non-keyword based search is possible. We now describe each problem in more detail:

### A. Document Classification

This is the first action which is performed on a document entering the system. In this step, we assign a proper place to the document in our information model. A document might qualify to be put in two or more places in the model. Once this step is over, the document becomes a leaf node in the directed acyclic graph which corresponds to activities. For document classification, we have to follow the following steps:

*1) Keyword Extraction:* When we get a document, we have to parse the document and extract its keywords based on the content. Additionally, we have to assign relevance scores to the keywords. The "relevance score" denotes the relevance of a given keyword to the document. These keywords and relevance scores are later used to relate one document to another.

*2) Inserting Documents in The Activity Model:* In this step, we have to decide the leaf node(s) where this document should go in the activity model. This decision should be based on the keywords as well as their relevance scores with respect to the given document. A document should be placed as the leaf node on a path only if all the nodes (ie. activities and sub-activities) on the path are relevant (atleast beyond a certain threshold) to the keywords of this document.

### B. Relevance Propagation

To support non-keyword based search, we must define the notion of two documents being related. Assuming such a relation exists between two documents, we can say that if a given word is related to the first document (or simply speaking, the given word is a keyword of the first document), then it would also be related to the second document. We define the calculation of the strength of relation between the given word and the second document as the first subproblem of relevance propagation. In other words, we have to find a suitable function to pass relevance of a given word from one document to another.

The other subproblem is to find an efficient way which can be used to calculate the relevance of documents to a given word over the whole activity graph. In essence, given a word, relevance propagation will assign relevance score to each document in the activity graph.

### C. Defining Activities and Sub-activities

Till now we have assumed that an activity graph exists. As stated above, an activity has a duration associated with it. It is very difficult for an automated system like ours to know when a new activity starts and when does an existing activity end. In other words, when a new document is entered into the system, our system has to decide whether it belongs to one or more of the current activities or whether it starts a new activity. The same is true for sub-activities.

This problem is somewhat simpler for information classes, since they are static. Our system will have many built-in information classes and hence would rarely face the problem of defining new information classes. We would also have certain information classes that are meant for a certain category of people eg. lawyers.

Although there would also be a mechanism by which the user would be able to define new information classes and activities, there should be a method to automatically infer

these.

In our project, we concentrate on the problem of relevance propagation, since this is basic to the realization of our system. In Section 9, we discuss various models for relevance propagation along with their advantages and shortcomings. In Section 10, we describe our current model. Since there has been a lot of research on document classification, we do not address this problem here. We assume that in future there would be such classifiers that would give the set of defining keywords along with their relevance scores for a given input document. Towards the end, we also discuss a few approaches that could be used for the activity - sub-activity definition problem. The ideas presented for this problem are currently in the conceptual stage and need some experimentation and research before they could be used as a viable solution for the problem.

## VI. Activities, Information Classes and a design rationale

This section describes the relation between information classes and activities, and briefly describes the rationale behind this design.

Information classes are designed to allow us to model what is commonly understood as data in a database system. It has a structured component, namely, the (recursive) schema definition of the class-subclass kind. In addition, it may have unstructured parts as well. On the other hand, activities are a special kind of information class, which specify a duration of the activity.

Consider the following simple example. Suppose that a very well-known author has written a book, that has many editions. The contents of each of those editions of the book is an instance of the 'Book' information class. The Book information class may have a large and pre-defined hierarchy of content, in the form of foreword, preface, table of contents, chapters, references, index etc. in various formats, namely, source (eg. tex, fig, word, etc.), printable (eg. ps or pdf), intermediate (eg. dvi). However, the Book information class is not an activity because the information in this class is independent of time (or, in other words, timeless!). In general, information classes represent timeless pieces of information.

On the other hand, the Book Writing information class can be an activity, strongly related to the specific edition of the book whose writing is being modelled. Writing a book is an activity, and includes correspondences with other authors, designated editors, proof-readers, versions of chapters, exercises etc. The process of book writing, and the information generated during this process is organized in this class. The Book Writing class may include the final version of the book as well (and may share it with the Book class). Queries about retrieving a specific correspondence with a co-author when a certain milestone was reached is answered by this class.

In our opinion, one of the main mechanisms used by human beings for correlating events is temporal proximity. For example, a professor may remember a discussion with a co-author on self-organizing information systems while at Toronto. The role of Toronto is completely secondary to the issue of self-organizing information systems; it however serves as a temporal key to identify that piece of information. Queries of the kind: find X when I was doing Y, seem to be particularly relevant for personal information systems. We therefore believe that there should be a significant role allocated to the notion of activity, which has the notion of duration built in as a primary feature. Together with the notions of an activity hierarchy, the concept of activity offers an approach to an event modelling mechanism that more closely matches the way we interact with the world.

## VII. Retrieval of Information in a Personal Information System

There are various types of queries that we intend to support in our system. A query could consist of just a number of words that describe the document that we are seeking. A query might also contain usual boolean operators such as 'and' and 'or'. This kind of querying is known as *content based querying*. Apart from this, a query may also have a structural part. A query must have at least one of the two parts (it can have both the parts simultaneously). The structural part is similar to traditional query languages, where, the schema information is used by the query. We first consider the content based specification and then present the structural part.

### A. Content based querying

In its simplest form, a content based query is a collection of keywords. A slightly more complex form of a content based query constructs the query using more complicated boolean expressions over keywords. Finally, the use of the *WHEN* operator allows the specification of temporal joins in the query statement.

Consider the simplest form of a content based query, that is, the query is a conjunction of keywords. The problem faced by the system is to retrieve all information units, instances of information classes, activities and sub-activities that exhibit a high relevance to the set of keywords specified. Alternatively, the system retrieves the top-k information units, classes and activities, in terms of a certain relevance metric, where, k is a parameter. The parameter k may be constrained by the output device of the user, such as the screen size, bandwidth available to the user etc.

### B. Structural part of the query

The following examples illustrate the structural part of the query.

Example 1: Suppose the query pertains to an email:

$$\textit{Class} = \textit{Email} \text{ AND } \textit{Sender} = \text{Avi AND } \textit{Receiver} = \text{Jeffrey}$$

This can be simplified to:

$$\textit{E-mail Sender} = \text{Avi AND } \textit{Receiver} = \text{Jeffrey}$$

Example 2: Suppose the query pertains to a course:

Courses *TaughtIn* Fall 2003

to indicate that the user is interested in courses taught in Fall 2003. The structural component of the query is similar to standard query languages for structured data (e.g., relational, XML or object-oriented languages).

### C. Temporal joins

The use of temporal association in mapping information units seems to be a significant way by which human beings associate, map and recall information. In view of this technique, we introduce the *WHEN* join predicate, as illustrated by the following example.

.jpg WHEN *Activity* = CS719

The above query asks for all photographs taken when the activity with the name CS719 was in progress. The following query asks for all vacation activities taken while writing the paper "estimating frequency moments":

*Vacation* WHEN *PaperWriting* AND *Title* = "estimating frequency moments" AND *Author* CONTAINS "Avi" ?

## VIII. Relevance Propagation

We first look at the idea of Relevance Joins that is fundamental to relevance propagation. After this, we describe various models for relevance propagation.

### A. Relevance Joins: basic idea

We first consider the problem of retrieving all information units that exhibit a high relevance to the set of keywords. Conceptually the retrieval process works as follows. A relevance score is associated with each information unit in the system, that is initialized to 0. Recall that each information unit has associated with it a set of keywords together with their relevance scores. For each keyword in the list of keywords, and for each information unit in the system, if the keyword is present in the list of keywords associated with the information unit, then, the specified relevance of the keyword is added to the value of the relevance score for that information unit. More formally, let *A* be an information unit and *K* be a keyword. If *K* is a keyword appearing in the list of keywords of *A*, we denote by *r(A,K)* the score of how relevant the information unit *A* is for *K*. Let *r(Q,A)* denote a score of how relevant the information unit *A* is to the query *Q*. The initialization step is performed as follows.

$$r(Q, A) = \sum_{K \in Q} r(A, K)$$

Next, we define a *relevance join* operation and illustrate it with an example. Let *A* and *B* be two information units, for example, suppose that *A* is an article on processor scheduling and *B* is an article on memory management. Suppose that *A* has two keywords, namely, `processor scheduling`, with relevance 1 and `operating systems`, with relevance 0.5. Similarly, suppose that *B* has two keywords, namely, `memory management`, with relevance 1 and `operating systems`, with relevance 0.5. Suppose that the query was on the keyword processor scheduling. Since both *A* and *B* have a common keyword, namely, `operating systems`, the relevance join of *A* with *B* on the keyword `operating systems` gives *B* a relevance score as a function *r*, given as follows.

$$s(r(A, \text{processor scheduling}), r(A, \text{operating systems}), r(B, \text{operating systems})) \quad (1)$$

where, *s* is a certain composition function. The above function gives the relevance score of the information unit *B* upon navigation from the information unit *A*. *A* is called the anchor unit and *B* is called the destination unit of the join operation. The anchor keyword(s) for a join operation is the keyword in the anchor unit whose relevance has made the anchor unit relevant to the query. In this case, the anchor keyword is `processor scheduling`. The join keyword(s) is the (set of) keywords that are common to both units. In this example, the join keyword is `operating systems`. The destination keyword(s) are all the other keywords of the destination unit. In our simplest model, all the remaining keywords of the destination unit share the same join relevance as computed by equation (1).

A specific calculation for the function $r$ in equation (1) is as follows. Let the anchor unit be $A$, the destination unit be $B$, the anchor keyword in $A$ be $K$ and the join keyword be $J$. Let $r(Q, A, K)$ denote the relevance function of the keyword $K$ in the information unit $A$ for the query $Q$. Right now let us assume that we make a single word query. In this case, $Q = K$ and $r(Q, A, K) = r(A, K)$.

$$r_1(Q, A, B, J) = r(A, K) \cdot r(A, J) \cdot r(B, J) \cdot \alpha$$

$$r(Q, A, B) = \sum_J r(Q, A, B, J)$$

$r_1(Q, A, B, J)$ denotes the relevance of document $B$ to the query $Q$ on account of the join keyword $J$ and the document $A$. $r(Q, A, B)$ denotes the total relevance that is passed to the document $B$ by the document $A$ corresponding to the query $Q$.

The above equations make the implicit assumption that the relevance function $r$ is always less than 1. The constant $\alpha$ is chosen to be less than 1 so that navigation always reduces the relevance by at least a factor $\alpha$. This implies that applying relevance joins multiple times reduces the relevance geometrically (ie. the first level joins reduce relevance by at least $\alpha$, the next level reduces the relevance by at least $\alpha^2$, next by $\alpha^3$ and so on). Applying this to our example, with $\alpha$ = 0.9 we have,

$$r(\text{processor scheduling}, A) = 1.0$$

$$r(\text{processor scheduling}, A, B, \text{operating systems}) = 1 \cdot 0.5 \cdot 0.5 \cdot 0.9 = 0.225$$

## IX. Relevance Propagation Models

In this section, we present various mathematical models for the problem of calculating the relevance of information units for a given query (ie. a set of keywords) that were conceived by us. We also discuss the pros and cons of each model. This forms a foundation for the next section in which we define our current model for relevance propagation.

### *A. Eigenvector approach*

In this model, we want to ensure that only incremental relevances are passed on. This can be understood as follows. Suppose documents A and B are related to each other. Also suppose that A is related to document C and B is related to document D. Also, lets assume that in step 1, A and B get some relevance from sources other than C and D and pass part of it to each other. In step 2, A gets some more relevance due to C and B gets some more relevance due to D. However now, A passes a portion of only that relevance that it gets from C to B and similarly, B passes only a portion of that relevance that it gets from D to A.

In this model, we compute relevance as follows. Let M be an $n$ x $n$ square matrix, where, $n$ denotes the number of information units in the system. Thus, M has a row and a column for each information unit. Let A and B be two information units numbered $i_A$ and $i_B$ respectively. The entry M[$i_B$, $i_A$] gives a real number that is calculated as follows.

$$M[i_B, i_A] = \sum_{\substack{\text{Common} \\ \text{Keywords J}}} r(A, J).r(B, J) \tag{1}$$

That is, the joint relevance coefficient, M[$i_B$, $i_A$] is calculated as the sum of the products of the relevance values for each join keyword. The assumption is that each relevance value *r*(A, J) and *r*(B,J) is a real number between 0 and 1. Also, we initialize M[i, i] to 0. This guarantees the property of passing on only incremental relevances.

Let $\lambda$ be an $n$-dimensional column vector that is initialized as follows based on the query Q.

$$\lambda_{i_A}^{(0)} = \sum_{\substack{\text{K} \in \text{Q} \\ \text{K} \in \text{Keyword(A)}}} r(A, K), \quad 1 \leq i_A \leq n \tag{2}$$

The term $\lambda_{i_A}^{(0)}$ calculates the relevance of the information unit A based on the query Q using the relevance information directly available and before performing relevance joins. The iterative step is given by the following equation,

$$\lambda^{(i+1)} := M\lambda^{(i)} \tag{3}$$

Since, the relevance values should lie between 0 and 1, we normalize $\lambda^{(j)}$, at the end of each iteration j, j = 0, 1, ..., as follows.

$$\lambda^{(j)} := \frac{1}{max_{r=1}^{n} \lambda_r^{(j)}} \lambda^{(j)} \tag{4}$$

$$\text{if } max_{r=1}^{n} \lambda_r^{(j)} \geq 1$$

This method is exactly analogous to the power method for finding the dominant eigenvector of a matrix and hence it converges if $\lambda_2$ / $\lambda_1 < 1$.

We calculate the relevance vector as

$$\lambda = \lambda^{(0)} + \lambda^{(1)} + \dots$$

The problem with this approach is that we cannot say for certain that the quantity that we will get at the end will indeed be a ranking of the documents in terms of relevance. Also, although we know that the power method for computing the dominant eigenvector converges, its speed depends upon the ratio of the first eigenvalue to the second eigenvalue. Since we cannot say anything about this ratio, we cannot say anything about the rate of convergence or the computation required for this method.

### *B. All Paths Approach*

In this model, we first define the paths along which relevance propagation must take place. We then present the various computation models for this approach which we tried but found to be unsatisfactory for tackling the problem.

For this model, we consider a document-keyword graph as follows. Every document corresponds to a single circle node in the graph. Every keyword corresponds to a square node in the graph. If a keyword is in the list of keywords corresponding to a document, then there is an edge between the document node and the keyword node.

For us, a relevance passing path is any path between two documents that uses a single keyword only once. In other words, all the square nodes (that correspond to keywords) occuring in the path should be unique and there should be no repetition. Note that a single document might appear multiple times in the path since we are not imposing any condition on the circle nodes (that correspond to documents).

At the start of relevance passing, only nodes which have the query keyword in their keyword-list have any relevance scores with respect to this query. We call these nodes "source nodes". In the first step of propagation, the source nodes will pass relevance to nodes which are directly related to them ie. the two documents contain some common keyword (which is not the query keyword). This will be done by using the relevance passing function defined in section 8A. In terms of the graph described above, this refers to those paths that originate from source nodes, contain only one square node (that does not correspond to the query keyword) and terminate at a circle node. The new nodes which have now gained relevance are added to the pool of source nodes and these new source nodes now pass relevance to other nodes related to them through common keywords (other than the ones through which they gained relevance).

It should be noted that the relevance scores that we use for passing relevance at each node in the graph corresponds to the original relevance of a node with respect to a keyword. We do not consider the new relevance scores which might be acquired as relevance passing progresses. This is different from the eigenvector approach above. However, in this model,

as stated above, if two documents contain two or more common keyowrds, then relevance passing between them occurs multiple times (because of different paths).

To calculate the relevance of a node to a given query keyword, we consider all paths which start from a source node and end at this node. Therefore, the relevance of this node (and hence the document to which it corresponds) to the given query can be defined as the sum of the relevance scores this node will get using all such possible paths.

Though, this model captures the basic notion of what relevance is, it suffers from serious implementation issues. The all paths approach as such involves calculation of all possible paths over the graph, which in itself is computationally complex (since there could be an exponential number of paths in general). This makes the addition and deletion of nodes computationally infeasible. A good algorithm for this approach should be incremental in nature. By incremental, we mean that on doing an addition or deletion of a node, or on changing the relevance of a keyword to a node, we should be able to propagate the changes without doing computations over the whole graph again.

*1) Normalization Model:* In this model, we maintain two matrices. First is a $N$ x $N$ matrix which will be used to store the total relevance passed from node $N_1$ to another node $N_2$. The second matrix is a $N$ x $W$ normalization matrix. The need for this second matrix is explained below.

In the all paths approach, we consider only those paths which do not contain the query keyword as an edge. This makes the results query specific. This also creates a problem that the relevance passed from one node to another is keyword specific. In this approach, our main matrix stores the total relevance passed between nodes using all the keywords. To incorporate keyword exclusion, either we can add one more dimension $W$ to the first matrix, and then an entry in the matrix will store the relevance passed from $N_1$ to $N_2$ for a given keyword $W$ using the all paths algorithm over a graph that is similar to the above graph except that it doesn't have the node corresponding to $W$. Instead, we choose to use another matrix which we call the normalization matrix. This matrix stores the factor by which we need to normalize the total relevance passed to get the specific relevance for this keyword.

To calculate relevance of a node to a given keyword, we take the entries in the column for this node in the first matrix, suitably modify these values according to the normalization matrix and then add them up.

The problem is that on addition or deletion of a node we will have to recompute the first matrix which is computationally infeasible for each addition/deletion. Also calculating the normalization matrix exactly is infeasible and we can only approximate it.

*2) Differential Model:* In this model, we use a $N$ x $N$ x ($N$ x $W$) matrix, where $N$ is the number of nodes and ($N$ x $W$) is the list of keywords for each node. An entry ($N_1$, $N_2$, $N_3$, $W_3$ in this 4-dimensional matrix represents the change in the relevance passed from $N_1$ to $N_2$ when there is a change in the relevance of $W_3$ with respect to the node $N_3$. This model assumes that a linear function is used to pass the relevance from one node to another. Using this model, it will be very easy to propagate changes when only the relevance scores change. The biggest problem with this model is that effectively we will have to calculate the whole matrix every time an addition or deletion of a node takes place. Also, it requires a huge amount of storage space.

## X. Current Approach

In this section, we describe our current relevance propagation model. As stated earlier, we assume that an average user looks at only the top 20-30 items. Therefore, we want an approach by which we can compute the top 50 or so results quickly. Also, we want the insertion, modification and deletion of files in the system to be efficient. We want to avoid the huge computation required in the previously stated models for these actions, since these might be quite frequent.

We form a graph comprising of all the documents in the system. Two documents are connected if they have a common keyword. An edge in the graph means that relevance can be propagated between the two documents that are connected by the edge. As can be seen, this graph is an undirected graph. For the purpose of calculating the relevance propagation between documents we do not need the edges of this graph to be given any weights. However, as will be explained later (in Section 11A), we might need to assign certain weights to the edges in this graph if we want to come up with Activity definitions (one of the problems mentioned in Section 5) according to the approach discussed in Section 11A.

In this approach, for a given query, we always maintain the top $c \cdot k$ documents (in terms of relevance) at each step in the discovery process of documents. $k$ is a parameter (around 30-40) that represents the number of documents that we expect the user to go through in response to her query. $c$ is a small number (around 2-3). As will be explained in Section 12A, we will be giving the final ranked list of documents using rank aggregation of two lists (one of them being this list and the other being the ranked list output by the plain keyword-based search module). We output a list of $c \cdot k$ elements from each module, even though we present only the top-$k$ documents to the user, so that we do not miss out on any relevant document and the final top-$k$ elements output by us are the $k$ most relevant documents for the query. We break the algorithm into three steps as follows:

- Initialization Step: Whenever we are given a query, we first find the documents that directly contain the query keyword(s) in their keyword list. We create a list of $c \cdot k$ elements and add all these documents into the list, if there are less than or equal to $c \cdot k$ such documents. If there are more documents than what the list can contain, we add only the top $c \cdot k$ documents (in terms of relevance) to the list. The documents in the list after this step are the documents from which we start the search for other relevant documents that do not contain the query keyword(s).

- Iteration Step: Using the document graph, look at all the documents that are reachable from the documents in the current $c \cdot k$ list. A document might be reachable from multiple documents in the list. After identifying all the documents, assign relevance to each one of them by using the relevance propagation function described in Section 8. If a document is reachable from multiple documents, it will get relevance due to each one of them and its net relevance will be the sum of all these individual relevances.

  Now look at the documents in the $c \cdot k$ list. If there is a document in the list that has relevance lower than any of the documents discovered in this iteration step, then it will be evicted from the list and the newly discovered item will be added to the list. In other words, we now consider all the documents in the current $c \cdot k$ list and the newly discovered documents together and populate the $c \cdot k$ list with the top $c \cdot k$ documents (in terms of relevance) out of these documents.
- Termination Step: If after one step of the iteration the $c \cdot k$ list remains unchanged, ie. no new documents are added to the list, we terminate our search. This is because any more iteration steps will not cause any change in this list of documents. Also, if at any step in the iteration, a document that is added to the $c \cdot k$ list has relevance less than a certain threshold, we stop the algorithm and output the current $c \cdot k$ list along with the relevance scores of documents in this list. This second condition is imposed to both speed by the algorithm and to avoid documents that are very marginally related to the query.

The above steps can be carried out quite efficiently by designing the data structures carefully (and/or by using insertion sort or some other sorting algorithm etc.). Therefore, answering queries using this model is a simple computation and can be done quite efficiently.

### A. Document insertion, modification and deletion

Since we are doing all computation on the fly using the graph of all the documents in the system, document insertion, modification and deletion are all accomplished very easily and efficiently. All we have to do is to delete some links in the existing graph and add/or some new links. So when a new document is added to the system, we create a node representing the document in the graph. We now look at all the keywords in its keyword list and create an edge between this document and any other document that contains even one of these keywords in its keyword list. Upon deletion of a document, we simply remove the node corresponding to it from the graph along with all the edges linked to it. When a document is modified, it is equivalent to deleting the old document from the system and adding a new document to the system. Therefore, to summarize, this approach can handle document insertion, modification and deletion very efficiently.

### B. Advantages of this model

This model requires us to store very little information. Specifically, we only need to store the graph corresponding to all the documents. Moreover, it is very easy to handle changes in the system (document insertion, modification and deletion). The model is also quite computationally efficient.

### C. Rationale behind the model

In this model, at every step, we are storing only the top $c \cdot k$ items that are relevant to the query. Moreover, the document discovery process is designed so that only documents reachable from these documents are considered. Intuitively, this means that we consider only those documents to be relevant to the query, which get their relevance because of other highly relevant documents. We are ruling out those documents that accumulate relevance because they are connected to lots of other documents in the graph which might be only moderately or minimally related to the query. This will be the case if there is a document which is tagged by general/common keywords and it is some kind of a general reference. eg. a book on operating systems will be related to almost any query related to computer science, although the relevance might be more or less depending on the query. We assume that a user is usually looking for only a specific document or a few number of documents that are quite specific to the query. Every document will have certain specific features and certain general features. We expect the user to use the specific features of the document when she is searching for it. So, going back to our example, if the user is indeed looking for the book on operating systems, she should probably write the name of the book or the name of the authors or some such specific thing.

## XI. Some approaches for the activity definition problem

Here we present some of the possible approaches for the problem of coming up with activities and sub-activities (defined in Section 5). As mentioned there, there are just some ideas that seem quite promising. These ideas need some amount of experimentation and analysis before they can be used.

### A. Clustering

In this approach, we make use of the graph consisting of all the documents that is described in Section 10 above. In addition, we give weights to all the edges in the graph. The weight is like a distance measure between the documents connected by the edge. Documents are at a distance of 0 if they have exactly the same keywords with the same weights. If there is a high amount of relevance propagation between two documents, then the distance between them should be less. So, naively, we can consider the distance function to be 1 minus the relevance propagation between the two documents. By relevance propagation here we mean the function

$$\sum_J \alpha \cdot r(A, J) \cdot r(B, J)$$

Once we have this graph, we run a clustering algorithm over it. Our intuition is that we will get various clusters corresponding to various activities. Once we identify this top level clusters, we could go inside each cluster to identify sub-clusters that correspond to sub-activities. However, the biggest problem with this approach is that the clusters that we will identity will be non-overlapping whereas activities, by definition, could be overlapping (ie. two or more activities can contain the same document).

### B. Mining the relevance graph

In this approach also we make use of the document graph. We use techniques of data-mining on the graph to identify sets of keywords that have a lot of documents containing all or almost all of them. So if the keywords $w_1, w_2$ and $w_3$ are found in many documents and with quite similar relevances, it means that these keywords, along with the average of the associated relevance scores, represent an activity or sub-activity. Naively, if for a set $S$, the number of documents that are returned is large, then it is an activity and higher sets (ie. sets containing $S$ as a subset) are probably sub-activities of this activity. A technique like the Apriori algorithm might be used to output all such possible sets of keywords.

## XII. Full Content Indexing

Any desktop based search system should have support for keyword based search. This means that given a keyword, we should be able to retrieve documents which contain the given keyword. All current desktop search tools more or less provide only this capability.

Our system enhances the current search tools with beyond keyword (or non-keyword) based search, temporal joins and other features. All these extra features help our system to find documents that are relevant to the query produce a ranking of the documents. In addition to this, we also do a plain keyword based search and get another ranking of ducuments. To support keyword based search, we need to do full content indexing.

During full content indexing, we also assign a relevance score to every word with respect to the given document. For example, if a document contains only ten words, then for this particular document, each of these ten words will have an associated relevance score. These relevance scores are assigned based on various parameters like:

- Frequency: the number of times a given word occurs in the document. Naively, if a word occurs more number of times in a document, then that word is highly related to the document. However, the relevance should not be a linear function of frequency. Instead, there should be a threshold frequency, below which the relevance behaves as a linear function of frequency, but it tapers off above this threshold. This threshold might be a constant for all documents, or it might depend on the total number of words in the document.
- Capitalization: whether the word is in small letters or capital letters. A word in uppercase usually implies higher relevance.
- Relative Font Size: A higher font size as compared to the other words present in the document, signifies a higher relevance score.
- Bold/Italic: A word in bold form or italicized means that it is being emphasized upon and hence is quite relevant to the document in question.
- **Proximity:** For multi-word queries, a very important factor is how close do the query words occur in the document. The closer they are, the more relevant the document is to the query. eg. if I search for "operating systems", then a document that contains the two words (operating and systems) together is much more likely to be the one that I am searching for, as compared to a document that contains both the words separated from each other.
- Order: For multi-word queries, a document that contains all the query words in the order in which they appear in the query, is likely to be more relevant to the search as compared to a document that does not respect the order.

We have adopted the above parameters from Google's web search engine ([7]). These parameters are used by Google in both their web search as well as desktop search utility. Along with the above parameters, depending on specific formats, other parameters can also be used. For example, we can use the tags present in html pages (such as title etc.).

### A. Rank aggregation

Given a query, our system will compute two ranked lists of the documents. One ranked list will be obtained by doing a beyond keyword based search using the activity model and relevance propagation. Doing a keyword based search using full content indexing will give us another ranked list. Our system will aggregate these two lists and then present the combined result to the user. The problem is to meaningfully combine the two results. However, there are various well known and easily implementable algorithms for combining top k-lists ([8]). As stated previously, we assume that the user will not be looking at more than 20-30 search results, and hence rank aggregation will in itself take little time.

## XIII. Implementation Issues

The implementation of the automatic document classification and retrieval system, presented in the previous sections, can easily be divided in several well defined modules. As the aim of our system is to enhance the current desktop information organizing and search utilities, hence our system will support the features currently available on such systems.

One of the main features of such systems are format readers for different file types. The parser for our system will take its input in only a single format, namely XML ([9]). Since we use a modular approach in our implementation, we can easily build support for various file formats. We just need to write a module (or use an already existing module) for the required file format, which will convert the given document to our custom XML format. The XML data thus generated will be fed to

the parser, which will then parse it accordingly and generate a content based index.

Another feature is "up-to-the-moment accurate" search. Implementing this is also quite easy. We can capture the *write* system call or use some other technique to send a signal to our program (that is constantly running in the background) whenever a new file is saved or an existing file is modified or deleted. Depending on the event, we can take appropriate action, such as indexing a newly written file or deleting some index data when an already existing file is deleted.

Our system comprises of five modules which do the following:

- Full Content Indexing
- Document Classification
- Information Classes
- Activity Definition
- Relevance Propagation Mechanism

The first module is completely independent of the other modules. It is responsible for keyword based search. The next four modules handle automatic classification and beyond keyword based search. The first module has been implemented and is described in the next section.

The second module extracts keywords and assigns them relevance scores. But, as previously mentioned, we did not address this problem and also, there is no such engine available. Because of this, a working system based on our information model is not realizable currently. However, as soon as such a document classification engine is available, our system is easily implementable. A workaround approach to test the capability of our system is to manually tag some documents with keywords and relevance scores and give them as input to the other three modules. However manual tagging of documents is a time taking and strenuous job and because of the time constraints, we were not able to test our system using this approach.

## XIV. Current Implementation

Presently, we have built a small search engine which does full content indexing and retrieves file names given multi-word queries. Currently, it treats every file as a plain text file while doing the indexing. It ranks the results based on the frequency of occurrence and capitalization of the query word(s) in the document. While indexing, we also store proximity information, ie. how close two given words are. However, in the current implementation, we do not incorporate this in our ranking algorithm for multi-word queries.

The engine has a GUI front-end based on Qt. It lets the user to do a quick search as well as a full search. In quick search, we only search for the top 64 entries of each word. If the number of results is less than a given threshold, we promote the quick search to a full search.

In our implementation, we hash each word we encounter (using double hashing) and give every file we index a unique *docId*. The whole indexing mechanism works using four kinds of files:

### A. DocInfo Files

For every file we index, we have a corresponding docInfo file. It is currently used to store the proximity information in the form of offset from the start of the file. We also store the docId of the corresponding file, the number of words occurring in the file and the hashes of the words. This latter information is needed to update the index when a file is changed.

### B. Hash File

We have a single file in which hash value of each word is stored. Along with this, we also store the offset of this word in the index file (described below).

### C. DocId File

It is a single file used to store the docIds of the files present in the system. The docId assigned to a new file is the last docId assigned + 1. Also, if a file is updated, it is assigned a new docId. All the information corresponding to the earlier version of the file is removed and the modified file is treated like a new file.

### D. Index Files

We have 256 main index files, plus an index file for each word separately. The former ones are used to store the top 64 entries (by relevance) for that word, while the latter ones are used to store entries for all the files in which this word occurs. While searching, we initially search in the main index files only, unless the results are very less or the user demands a full listing. The entries in both the main index files and the subsidiary ones are sorted according to docIds. This helps in multi-word queries. Each entry contains docId, relevance and offset and the size of proximity information in the corresponding docInfo file.

## XV. Conclusion and Future Work

From the recent spate of tools released by all the major industry players, it is clear that the field of Desktop Search Systems is currently in focus. However, most of the work till now is quite trivial and does not create a personalized search experience for the user, which we believe, is essential for a desktop system. In our work, probably the first attempt in this direction, we present an information model and algorithms that enable efficient querying based on the model. We also incorporate all the features available in the other desktop search systems in town. The two most striking features of our model are its support for non-keyword based queries and temporal queries based on activities. In future, we envision that in response to a search query, we will get ranked lists of activities and information classes (based on relevance and temporal joins) in addition to a unified ranked list of documents. A user could click on any of the activities to find sub-activities and documents that belong to that activity and are relevant of the query.

There is a lot of work that can be done to enhance our system. Firstly, a document classification system or manually tagged corpus must be used to tag a large number of

documents and test the effectiveness of our system. Also, the approaches for activity - sub-activity definition, presented by us here, need to be experimented with and researched. Once all these things are in place, we hope that an actual system that incorporates all these features can be implemented efficiently.

## ACKNOWLEDGMENT

We would like to express our heartfelt gratitude to Prof. Sumit Ganguly without whose guidance, inspiration and constant motivation, this project would not have been possible. He introduced us to the problem and most of the ideas described in this paper were conceived during our meetings with him. The help extended by him in writing this report is invaluable.

## REFERENCES

[1] Google desktop search. [Online]. Available: http://desktop.google.com/about.html
[2] Yahoo desktop search. [Online]. Available: http://desktop.yahoo.com/features
[3] Apple spotlight. [Online]. Available: http://www.apple.com/macosx/tiger/spotlight.html
[4] Copernic desktop search. [Online]. Available: http://www.copernic.com/en/products/desktop-search/
[5] Haystack universal information client. [Online]. Available: http://haystack.lcs.mit.edu/
[6] Resource description format. [Online]. Available: http://www.w3.org/RDF/
[7] S. Brin and L. Page, "The anatomy of a large-scale hypertextual Web search engine," *Computer Networks and ISDN Systems*, vol. 30, 1998.
[8] R. Fagin, R. Kumar, and D. Sivakumar, "Comparing top k lists," in *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, 2003, pp. 28–36.
[9] Extensible markup language (xml). [Online]. Available: http://www.w3.org/XML/