UFMG - ICEx
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

UFMG
UNIVERSIDADE FEDERAL DE MINAS GERAIS

A Model for Web Mining Applications – Conceptual Model, Architecture, Implementation and Use Cases

RT.DCC.001/2008

ÁLVARO RODRIGUES PEREIRA JÚNIOR
RICARDO BAEZA-YATES
NIVIO ZIVIANI

FEVEREIRO
2008

A Model for Web Mining Applications – Conceptual Model, Architecture, Implementation and Use Cases

Álvaro Pereira
Dept. of Computer Science
Fed. Univ. of Minas Gerais
Belo Horizonte, Brazil
alvaro@dcc.ufmg.br

Ricardo Baeza-Yates
Yahoo! Research &
Barcelona Media
Barcelona, Spain
rbaeza@acm.org

Nivio Ziviani
Dept. of Computer Science
Fed. Univ. of Minas Gerais
Belo Horizonte, Brazil

Abstract

Web mining is a computation intensive task even after the mining tool itself has been developed. However, most mining software is developed ad-hoc and usually is not scalable nor reused for other mining tasks. This paper presents a Web mining model and implementation, referred to as WIM – Web Information Mining –, where rapid prototyping is possible.

The underlying conceptual model of WIM provides its users with a level of abstraction appropriate for prototyping and experimentation throughout the Web data mining task. Abstracting from the idiosyncrasies of raw Web data representations facilities the inherently iterative mining process. This paper details this conceptual model, together with its associated algebra, the architecture of the WIM tool, and its implementation. It also demonstrates how the model has been applied in several real Web data mining tasks. Resulting from this experimentation, WIM has proved to significantly facilitate Web mining prototyping.

1 Introduction

Data mining, and in particular Web data mining, is an iterative process in which prototyping plays an essential role in order to easily experiment with different alternatives, as well as incorporating the knowledge acquired during previous iterations of the process itself. Further, in order to facilitate prototyping, an appropriate level of abstraction must

be provided to the user or programmer carrying out the web data mining task. This paper presents such an abstract model.

Web mining applications use a combination of the following types of Web data [23, 9]: Web content, such as the text and URL of documents; Web structure, in the form of hyperlinks; and Web usage data, in the form of logs, with navigation or application specific data. We have developed a tool, referred to as WIM – Web Information Mining – with the goal of providing a simple high-level language for accessing and manipulating Web data, as well as data processing tasks [18, 35] for Web mining applications. This is not only a library of functions commonly used for Web mining, but a language which will allow the composition of operations and reuse of previous results within the mining process itself.

Web mining is not a closed world. This means that we cannot develop a tool for solving every Web mining application. During the design step of this work we studied a large set of Web mining techniques and the applications that these techniques could be employed on, aiming to delimit the scope of the WIM tool. WIM can be used for a series of data mining tasks, such as: association rules and sequential patterns mining; unsupervised learning tasks such as k-means clustering and clustering based on graph structure and text; search and text comparison; link analysis, with co-citation analysis and document relevance. WIM allows all these data mining techniques to be used together for the same application, integrating the data. WIM is designed to easily allow the addition of new tasks, wwhich can be done by a community of Web data miners interested to expand the WIM functionality.

In this paper we present the main issues involving the design and development of the tool and its underlying conceptual model with its main concepts, architecture, and language operators. Additionally, with example use cases we show how fast the implementation of a Web mining task solution might be when using WIM. From an industrial perspective, considerations for productivity, low learning curve, and short iterations time, are imperative for a given approach to succeed. For the first version of this tool, we focus on simplifying the development of Web mining prototypes rather than scalability. Considering the exploratory nature of Web mining, trying first a quick solution in a medium size data set allows to check if it is worth to develop an ad-hoc and scalable solution for the target problem.

We present five applications to which WIM has been used to implement solutions. The first application concerns a study on the evolution of content in the Web. The WIM program identifies parent documents, which are sources of copy, and child documents, which are more recent documents containing old content. Problems like duplication detection, URL comparison, association of a unique parent to each child, are all addressed with the Web evolution WIM program. We have implemented the same solution both ad-hoc [3] and through WIM, and we present a comparison between these solutions.

The second application concerns the manipulation of search engine usage logs in order to implement a document usage-based relevance weight, with the general objective of improving ranking for queries. For this application we used a Yahoo! search engine log with 22 million clicks, and imported the real pagerank from a Web dataset from United Kingdom, containing 77 million entries.

The other applications are: i) a comparative study of linkage evolution between new

pages with new content and new pages with old content, ii) the manipulation of query logs, in order to identify a series of properties for each distinct query that occurs in the log, with the objective of studying search engine user intent, and iii) composing a pool of documents for relevance assessment, based on different retrieval methods.

The remainder of the paper is organised as follows. The next Section overviews previous work related to the goals of this paper. Section 3 details the conceptual model underlying the WIM tool. Then the set of operations that WIM provides in order to manipulate and mine web data are summarised in Section 4. Section 5 outlines the architecture used in the current implementation of WIM. Further implementation details are discussed in Section 6. Section 7 demonstrates with practical applications how WIM can be used to mine information from real web data, and Section 8 present our conclusions.

2 Related Work

There are two sets of systems that have some similarity to WIM. The first set represents data mining frameworks, that although are not specially designed for Web data, may be able to cover some Web mining applications. The second set represents the SQL-like query languages for Web data, whose similarity is to deal with Web data.

In the set of the data mining frameworks, some commercial SQL databases have add-ons for data mining modules, as Microsoft [28], Oracle [29] and IBM [27]. They have business-driven solutions that are not related to Web data. Weka [35] is a very known data mining framework, which implements a collection of machine learning algorithms for data mining tasks. The algorithms can either be applied directly to a dataset or called from a Java code.

In the set of the SQL-like query languages for Web data, we highlight Whoweda [30, 5, 6], WebSQL [26], WebOQL [1] and StruQL [16]. They are query languages and model the Web as a kind of relational table, allowing more sophisticated queries and operations than the keyword-based query on the text of the document, as search engines provide. They also take advantage of the Web as a graph, allowing association among documents. An agent based system for Web mining is described in [19], but its goals are different as there is no formal mining language.

WebBase [32, 10] is an important Web warehouse project which also incorporates a query language, whose main goals are to manage very large collections of Web pages, to enable large-scale Web-related research, and to locally provide a significant portion of the Web. They view a Web warehouse simultaneously as a collection of Web documents, as a navigable directed graph, and as a set of relational tables storing Web pages properties. Thus, the Web warehouse is modeled as a collection of pages and links, with associated page and link attributes. The model incorporates the ranking of pages and links.

Although we also address the problems of Web data access, manipulation and retrieval, our focus is on the ability of the model in prototyping Web mining solutions, which is not covered in the previously referred works. In this sense, the WIM model language is in a higher level compared to other languages. To the best of our knowledge, WIM is the first

system for fast prototyping of Web mining applications.

3 Data Model for Web Mining

Most Web mining algorithms and software currently available have been developed ad-hoc, with specific data structures in mind. These data structures are used to represent the Web data to be mined. As described in the previous Section, three different types of Web data are usually combined to different degrees for mining purposes, i.e Web content, Web structure, and Web usage. A large variety of different data structures and formats are used by existing solutions to store and access all required sources of data, typically graphs, text documents, and relational tables, together with associated indexing structures.

While this approach has certainly led to very significant advances in Web data mining, it also poses severe limitations to current research. On one hand, this diversity limits the readily reuse of existing tools and algorithms by other researchers, unless an open source approach is taken, not currently a very widespread choice. On the other hand, even more importantly, it also hampers the direct comparison of existing research contributions.

A more abstract model is, therefore, required in order to overcome these limitations. Inspired by some of the same principles that motivated the advent of the relational model decades ago [11], a simple but powerful data model is described that will allow to represent all data needed for Web mining purposes. The main goal is to provide a layer of data independence from the raw data that is to be mined. This Section describes the data structures used to provide an appropriate abstract view of Web data used for mining. Then, Section 4 describes the set of data manipulation operations needed in order to address a large variety of Web mining tasks.

3.1 Preliminary Definitions

This subsection provides basic definitions used in order to describe the data model of Section 3.2. They are all taken from the standard relational databases literature [36].

Assume there is a set of *attributes* $\mathcal{U}$, each with an associated domain[1]. A *relation schema* R over $\mathcal{U}$ is a subset of $\mathcal{U}$. A *database schema* $D = \{R_1, \ldots, R_n\}$ is a set of relation schemas. A *relation* r over the relation schema R is a set of R-tuples, where a *R-tuple* is a mapping from the attributes of R to their domains.

The value of an attribute, $A \in \mathcal{U}$, for a particular R-tuple, t_1, is denoted $t_1(A)$. Such a value will be referred to in this model as *element*.

3.2 A Relational View of Web Data

The model presented in this paper, referred to as WIM (Web Information Mining Model), proposes that the three types of existing Web mining data (Web content, Web structure

[1] As will be described in Section 6, only two domains will be allowed: floating points and strings. An exception is for attributes that represent identifiers, which are integers.

and Web usage data) can all be represented solely by *relations*, as defined above. Further, only two types of relations will be needed: namely *node* relations and *link* relations.

Node relations model nodes in general, such as pages of a Web graph, terms of a document, or queries or sessions of a query log; and *link relations* model edges (links) of a graph, such as links of a Web graph, word distance among terms of a document, similarity among queries or time overlap among sessions of a query log.

Figure 1 includes specific examples of how the concepts of *node* and *link* relations can be used to model real Web data. Relations termed *sn0*, *sn1PR* and *snPR2*, for example, are all node relations which represent Web documents. A *node* relation is characterized by its identifying attribute, which identifies the Web document. It may also include many other attributes such as URL and title, as is the case for these three relations. Additional attributes can be added to relations as needed by the mining task at hand, such as attributes referring to a document's page rank, as shown by attribute *pr* in relations *sn1PR* and *snPR2* of Figure 1.

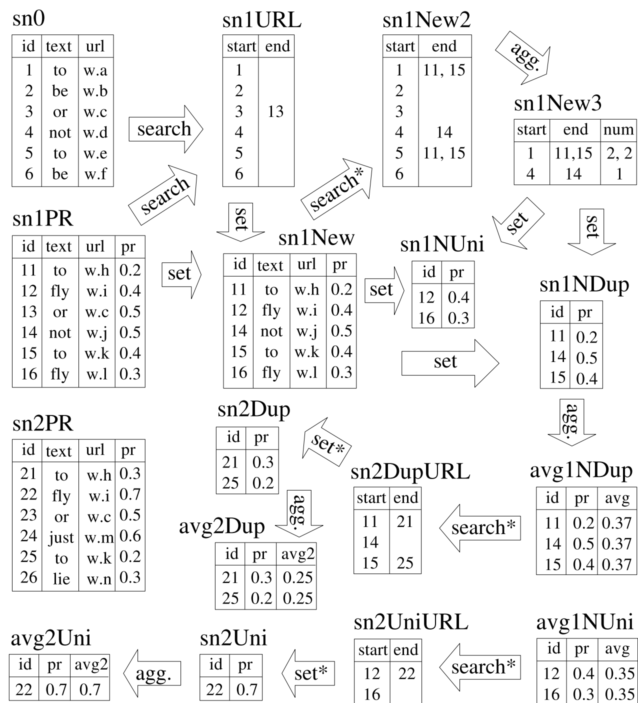

Figure 1: Example of a WIM program to study the average pagerank evolution for duplicated and new-content pages.

Any *link* relation, like *sn1URL*, *sn1New2*, and *sn1New2* in this Figure, must have attributes representing *start* and *end* nodes, which are represented on a different *node* relation. Thus, *start* and *end* attributes are foreign key attributes to their associated node relations, which need not always be the same relation for both attributes. Attribute *end* can contain a list of values[2], which is the list of nodes to which each start node links (this

[2]The authors note that this paper does not address issues like *normalization* of relations, but the focus

list can be empty). In addition, this relation may also include another attribute for the *anchor* text of a link between Web documents.

Similarly, relation termed *click* in Figure 2, which is another example of how relations can be used to model real Web data, is also a *node* relation. It represents Web usage data. In this relation, every action of the user is represented by a different tuple. One attribute identifies the session, another attribute identifies the query, and a third attribute identifies the clicked document number.

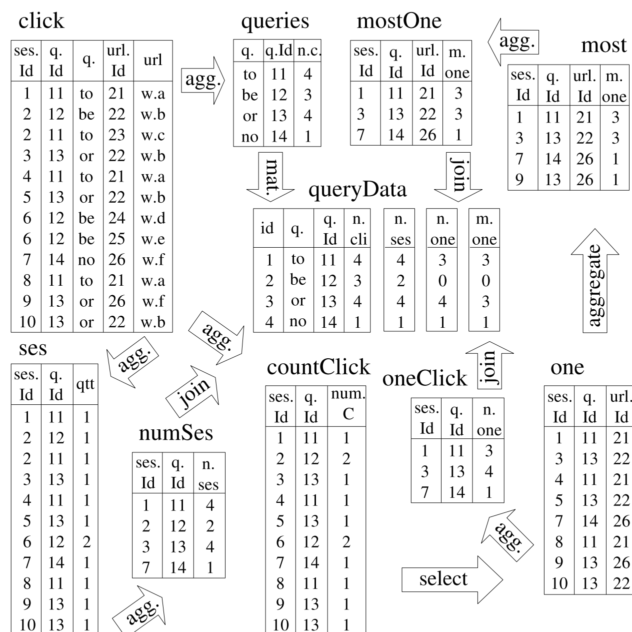

Figure 2: Example of a WIM program to manipulate usage data.

Every WIM relation must have an attribute to store the identifier of tuples, which will be the primary key of that relation. For simplicity, and without loss of generality, this attribute is always named ‘id’ for node relations and ’start’ for link relations.

The term *compatibility* will be used throughout this paper in order to refer to a pair of relations associated by a foreign key constraint. In the examples above, relation *sn1URL* of Figure 1 is said to be *compatible* with *sn0* as well as *sn1PR*, since attribute start is a foreign key to *sn0*, and attribute end contains values which are foreign keys to *sn1PR*.

The examples above model all three types of data used for Web mining: content, structure and usage. Note that normally there is a link relation associated with a node relation. Typically, a node relation models Web content data represented by documents, whereas the link relation models the structure among those documents. Usage data is modeled as a node relation. Typically, it is also associated with another node relation that represents the documents, given that usage data has data about documents clicked on by the user.

is on the data model itself. In any case, as will be justified in Section 6, no update operations need to be considered on node or link relations, therefore normalization will probably never need to be addressed.

Despite its apparent simplicity, the WIM model presented here suffices to model the most common types of Web data need for mining purposes. Web data bases are certainly task and user dependent. As will be illustrated in Section 7 with specific examples, however, any Web collection can be viewed as a set of appropriate node and/or link relations, as defined above, and then be mined using the set of operators described in Section 4. WIM represents, therefore, an appropriate level of abstraction that could be used as a unifying Web data mining framework.

Node and link relations are manipulated using a well-defined set of operators, referred to collectively as the WIM algebra, in order to mine the raw data they abstractly represent. This algebra is detailed in the following Section.

4 WIM Algebra

Before presenting the WIM algebra, we introduce some concepts regarding the use of the operators.

A WIM *program* is a sequence of operations applied to relations. Actually Figures 1 and 2 are examples of WIM programs, where arrows represent operations. We use these programs in the following sections to explain the operators of the WIM algebra. The WIM modeling language is a kind of dataflow programming language, which is derived from the functional programming paradigm.

A *temporary relation* is an output of an operation, which can be used anywhere in the current WIM program, but not in other programs. For that, users need to materialize a temporary relation, by using the materialize operator presented in Section 4.7.

The concept of *compatibility* presented in Section 3 is not limited to given pairs of node-link relations. Relations originated from the same physical relation are said to be *compatible* among themselves and with respect to the original physical relation, and inherit attributes of other compatible relations. WIM algebra takes a high profit of this property, not only for dealing with views and avoiding replication of attributes, but also to eliminate the need of an operator to project attributes, given that any relation can consider that any attribute of other compatible relation is also an attribute of itself.

Notice that for link relations, the relation to which the start nodes are compatible may differ from the relation to which the end nodes are compatible. Furthermore, a link relation may be compatible to a node relation, which allows some operators to use attributes of a compatible node relation when dealing only with a link relation as input.

The WIM algebra currently has fifteen operators. They are divided in two categories: data manipulation operators, which are presented from Section 4.1 to Section 4.7, and mining operators, which are presented from Section 4.8 to Section 4.15. Some control operators, as an operator for showing a relation to the user, are ommited due to their simplicity.

4.1 Select

This operator selects tuples from the input relation, according to a condition applied to a numeric attribute. The conditions are: equal, different, greater than, greater than or equal to, less than, and less than or equal to. Three options are available: *Value*, *Attribute*, and *Top*.

For option Value, elements of a numeric attribute are compared against a given value passed by the user. For option Attribute, the comparison is performed between elements of two different attributes of the same relation. For option Top, the conditions for comparison presented above are not used. Only a given number of elements with the highest or lowest values are returned.

Figure 2 presents an example of the select operator, with option Value. Only tuples whose value in attribute 'num.C' of relation countClick is equal to 1 are returned in relation 'one'. Select can also be applied to link relations. For instance, in Figure 3, which is another application of the WIM that is studied in Section 7, the select operator is applied twice, for relation tfidf2 and relation normPR2. In both cases, option Top is used, to return the two greatest values in attributes 'tf2' and 'tfpr', respectively, which represent labels for links from start to end nodes.

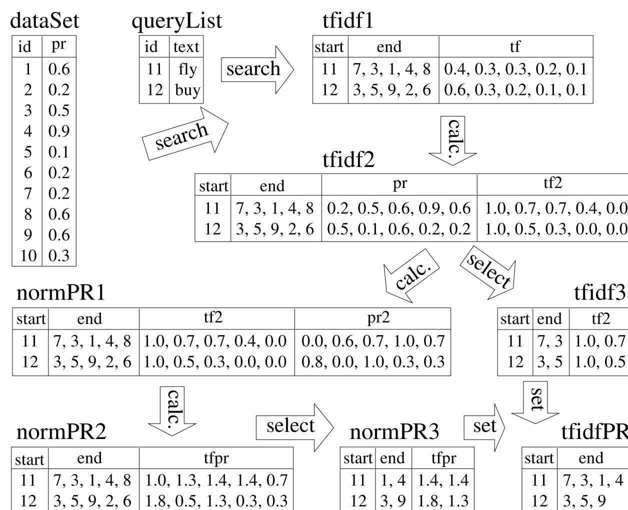

Figure 3: Example of a WIM program to compose a pool of documents for given queries.

As an example of the select operator with option Attribute, observe the relation relUsDocs in the center of Figure 4, which represents another example of a WIM program. For option Attribute and condition greater than, applied to attributes 'pr' and 'u.pr', only the tuples identified by 1 and 2 would be returned, because 0.3 and 0.5 are, respectively, greater than 0.1 and 0.2.

Select can also be applied to link relations. For instance, observe the other WIM example of Figure 5. Operator Select is applied to the link relation relSearchUrl, where the attribute 'sim' represents label for links from start to end nodes. Relation relSeDifUrl

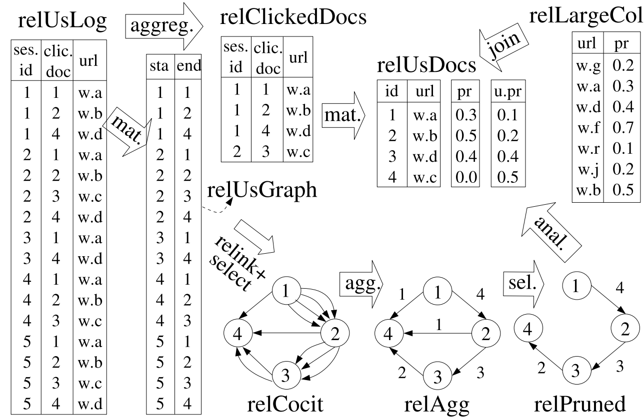

Figure 4: Example of a WIM program to study the usage pagerank.

contains only links whose 'sim' value is 0. Select is also applied to the link relation relAgg in Figure 4, returning only the links whose label is greater than 1, which are represented in relation relPruned.

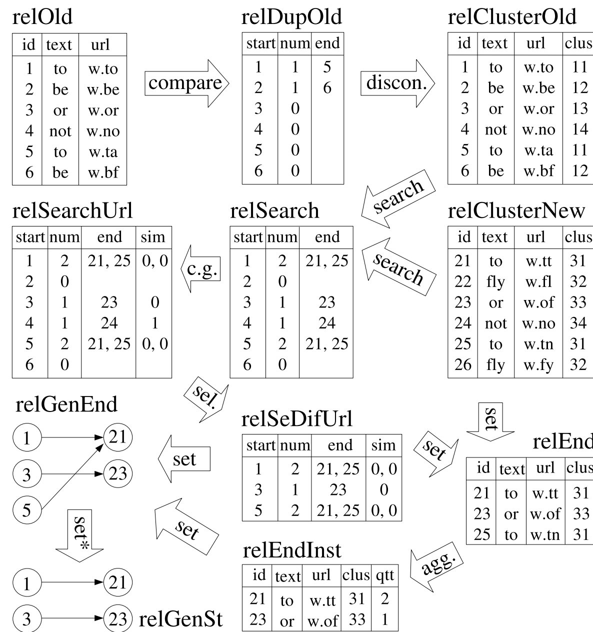

Figure 5: Example of a WIM program to study the Web content evolution.

4.2 Calculate

This operator is used for mathematical and statistical calculations for numeric attributes. Although the permitted operations differ for each option (for example, division is not applicable for a list of values), the following operations are available for at least one of

the options: sum, difference, multiplication, division, rest of division, average, standard deviation, percentage and normalization.

Both node and link relations are accepted, and two options are available: *Constant* and *Pair*. For option Constant, the calculation is performed between a constant value passed by the user and each element of an attribute. For option Pair, the calculation is performed between the corresponding elements of two attributes, in the same tuple.

In any case a new attribute is added to the output relation, to store the result of the calculation. The calculate operator is applied to relation tfidf1 in Figure 3, with option Constant, to normalize values between 0 and 1 in attribute 'tf', returning attribute 'tf2' in relation tfidf2. In the same figure, calculate is used in relation normPR1, with the Pair option, to sum the corresponding values in attributes 'tf2' and 'pr2', returned as attribute 'tfpr' in relation normPR2.

4.3 CalcGraph

The CalcGraph operator applies one of the calculation operations presented in Section 4.2 for every link of a graph, so that the calculation is applied in pair of start and end nodes. The input is a link relation, whose values for the start and end nodes are taken from their compatible node relations. The users do not need to pass the compatible node relation, once the WIM is able to recover this metadata. However, the users must inform to which attributes for both start and end nodes the operator is applied.

The output is a link relation compatible with the input, with a new attribute to store the result of the calculation. For example, consider the relation relGenSt graphically represented in the bottom of Figure 5, whose compatible node relation for the start node is relation relClusterOld and for the end node is relation relClusterNew. Applying the sum operation for attribute 'clus' of both compatible relations would result in 42 $(11+31)$ and 46 $(13+33)$, respectively labels for links from 1 to 21 and from 3 to 23 in relation relGenSt.

4.4 Aggregate

The aggregate operator is used to combine values of a given attribute of a relation. There are two options: *Single*, which does not imply in deleting tuples, and *Grouping*, which implies in deleting tuples. In both cases one of the following operations is applied to values of a chosen attribute: sum, average, counting the number of elements, maximum, minimum, standard deviation, geometric average, mode or median.

Option Single means that an operation is applied to all values of a single attribute. In this case the result is a single value, although, in order to guarantee input/output uniformity among operators, we represent the output as a new attribute, with the same value for every node. The aggregate operator with option Single is applied to attribute 'pr' of relation sn2Dup in Figure 1. A new attribute, 'avg2', is added in relation avg2Dup to store the average of values in attribute 'pr'.

For option Grouping, sets of tuples with the same value replicated in an attribute are removed. Only different values are kept. Another attribute may be used to have one of

the available operations applied to its values. A new attribute is included in the output to store the result of the operation.

For example, the aggregate operator is applied to the 'clus' attribute of relation relEnd, in the bottom of Figure 5. The operation results in relation relEndInst, where only one of the two tuples with the value 31 in attribute 'clus' is kept, and a new attribute counts the number of tuples with each value. The aggregate operator with option Grouping is also applied to the link relation relCocit in Figure 4, which the number of links is counted and returned as an attribute of the output link relation relAgg.

The program in Figure 2 has several examples of the aggregate operator with option Grouping. For instance, aggregate is applied to relation 'one', with option for counting the number of elements, to attributes 'q.Id' and 'url.Id'. It means that any different pair of values for these attributes must be represented in the output, which is relation 'most', where attribute 'm.one' stores the number of tuples with the same pair of values. For instance, the pair (11, 21), the first entry in 'most', occurs three times in relation 'one'.

4.5 Set

This operator is used for *intersection*, *union* or *difference* of tuples in two different relations. The user must choose the attribute to be compared for each input relation, which is not necessarily the identifier attribute.

For the intersection, the output is compatible to both input relations, whereas for the difference the output is compatible to the first input. For the union, if both inputs are compatible to the same relation, the output will also be compatible to that relation.

The set operator allows inputs of different types, i.e., a node relation and a link relation. In this case, the user must indicate which attribute of the link relation must be considered: the start or the end attribute. The type of output is inherited from the first input indicated.

For example, observe the Set operator applied to relations relClusterNew and relSeDifUrl in Figure 5. Consider that the first input passed is relClusterNew, and the operation is intersection. Attribute 'id' is used from relClusterNew and attribute 'end' is used from relSeDifUrl. The result is relation relEnd, which contains a subset of tuples from relClusterNew, identified by 21, 23 and 25.

Figure 1 presents various examples of the set operator. In every case, it is applied to the start or end node of link relations, together with a node relation. For instance, set with option difference is applied to attribute 'id' of relation sn1New and for attribute end of relation sn1New3. In Figure 3, set is applied to the link, which means start and end node pairs, for relations tfidf3 and normPR3, with option union. The result is the union of the graphs, in relation tfidfPR.

4.6 Join

The join operator is used to add an external attribute in a given relation. This operation is different of the set operation, which returns only the identifier of an input. The join operator receives two relations, in which three attributes must be passed. The first two

attributes are used for comparison, and each one belongs to one different relation. The third attribute belongs to the second input relation, and its values must be joined into the first input relation in order to generate the output.

A default value must be passed in order to be used when a value in the first relation is not found in the second relation. When a value in the first relation is found more than once in the second relation, only one value from the second relation is joined, which is the first value found.

In Figure 4, join is applied to relations relUsDocs and relLargeCol, for joining attribute 'pr' (which previously did not exist in relation relUsDocs) according to the values of attribute 'url' in both inputs. The output is represented in the same relation relUsDocs, where 'pr' is a new attribute, which is the result of materializing the new attribute in the relation.

Figure 2 has several join operations, which are used to add attributes of relations to relation queryData. This relation contains a set of attributes for each different query in a query log. For instance, attribute 'm.one' from relation mostOne is joined to relation queryData, using for comparison the attribute 'queryId' in each relation.

Similarly to the set operator, join also allows inputs of different types.

4.7 Materialize

Sometimes the output of programs must be materialized, not only to be used later without the need of running the program again, but also to allow users to dynamically create relations with their attributes. If the output relation is compatible to some existing previously materialized relation, an output attribute, which should have been generated by some operation in that program, is registered as a new attribute. Otherwise, in case the output relation is not compatible with other relations, the whole relation may be created with at least one attribute.

The two options for the materialize operator are *Relation* and *Attribute*. For the attribute option, users must pass a default value to be used in case an entry in the compatible target relation is not represented in the current attribute.

Figure 4 presents two examples of the materialize operator. First, attributes 'ses.id' and 'clic.cod' of relation relUsLog are used to materialize the link relation relUsGraph, as start and end nodes, respectively. Second, attribute 'url' of relClickedDocs is materialized in a new relation, relUsDocs. Note that the identifiers are automatically added and do not have any relationship with previous identifiers. According to the figure, the join and analyze operators return attributes to relation relUsDocs. However, this is just a representation in the figure, in fact both operators return other relations that have their attributes later materialized to relation relUsDocs.

Figure 2 also presents an example of this operator. Three attributes from relation 'queries' are materialized into relation queryData. Notice that there are three join operators in this figure, which are also adding attributes to relation queryData. If the users wanted these attributes to be managed later by another program, the materialize operator should have been used.

4.8 Search

This operator allows a textual attribute of a relation to be searched in a textual attribute of another relation. The typical application for this operator is querying, although it is also important for comparison of texts in general. The following comparison methods are allowed: conjunctive comparison, disjunctive comparison, TF-IDF, Okapi BM25, exactly the same text, and shingles [7]. According to each comparison method, different sub-options are available.

The output is a link relation whose start nodes represent tuples in the first input relation, where the queries are typically stored, and end nodes represent tuples in the second input relation, where the texts for searching are typically stored. Thus, the output is a bipartite graph, and users may take advantage of this property by using other operators in the sequence of their program, for example to manipulate only the list of documents returned, discarding the queries, which are represented by the start attribute of the link relation.

The order of the results of a query are implicitly represented by the order in which the end nodes appear. The end node at position zero for a given start node represents the top ranking document for that query. A new attribute is added to store the similarity between the query and the text.

Figure 5 presents an example of the search operator. Attribute 'text' of relation relClusterOld is searched in attribute 'text' of relation relClusterNew. The output is the relation relSearch. For instance, the text 'to' existent in tuples 1 and 5 in relClusterOld is found in tuples 21 and 25 in relClusterNew.

The search operator is also used in Figure 1 to compare the URLs of relations sn0 and sn1PR. The result is a graph with only one link, from 3 to 13, because only the URL 'w.c' exists in both relations. It is represented in relation sn1URL. The weight, that represents the similarity value, is not represented in relation sn1URL because it is not used later in this program. In Figure 3, two queries in relation queryList are searched in relation dataSet (attribute 'text' is omitted). The result is relation tfidf1, where a new attribute stores the similarity between the query text and each returned document, represented in the end node.

4.9 Compare

The compare operator permits the comparison among all the elements of a textual attribute of a single input node relation. The comparison methods are the same as for the search operator presented in Section 4.8. The output is a link relation with links among the documents that share a minimal similarity percentage informed by the user.

There are two options: *Sparse* and *Dense*. For the sparse option, the output does not have a new attribute, apart from the start and end nodes to represent the graph. A link is created between a pair of similar documents, in only one direction. For example, if documents A, B and C are similar, a link is created, for example, from A to B and to C, not among every pair of similar documents. Note that the similarity value cannot be

represented, because not all the links are represented (the representation seems a cluster of documents). For the dense option, every direction is represented in the output, which allows the addition of a new attribute to store the similarity value.

The compare operator is applied to the 'text' attribute of relation relOld in Figure 5, with option Sparse and comparison method "exactly the same text". Relation relDupOld shows that the text of tuples 1 and 5 ('to') and tuples 2 and 6 ('be') are the same.

4.10 CompGraph

Allowing the same operations as the search and compare operators, the compGraph operator receives a link relation and compares pairs of textual attributes of the start and end nodes. The values for the start and end nodes are taken from their compatible node relation, as occur to the calcGraph operator presented in Section 4.3. The user must inform which textual attributes for both start and end nodes are applied.

The output is a link relation compatible with the input, with a new attribute to store the result of the comparison, which is a similarity value.

Operator compGraph is applied to relation relSearch in Figure 5, for the comparison of the 'url' attributes of the compatible relations of start and end nodes, respectively relations relClusterOld and relClusterNew. The output is relation relSearchUrl, where the 'sim' attribute is added to store the similarity value for each link existent in relSearch. For instance, the link from 4 to 24 has similarity 1 because the URLs of tuple 4 in relClusterOld and tuple 24 in relClusterNew are the same.

4.11 Cluster

The cluster operator receives a node relation containing a set of attributes. Values of tuples in the passed attributes are used to produce clusters by using the k-means algorithm [24]. The number of clusters is also a parameter passed by the user. Both Euclidean and Manhattan distances are available. The output is a relation compatible to the input relation, with a new attribute to store the cluster number associated to each entry.

4.12 Disconnect

The disconnect operator allows the identification of clusters in a graph. The input is a link relation, and every node of the graph is classified into a cluster, according to its connectivity in the graph. The output is a node relation containing all the nodes of the graph, with an attribute to associate a cluster number to each entry.

There are two possibilities for this operator: to make clusters of connected components, in which the direction is not important, and clusters of strongly connected components [12], which takes into account the direction of a link.

Disconnect is applied to the link relation relDupOld in Figure 5, for clusters of connected components. As nodes 1 and 5, and nodes 2 and 6 are connected in relation relDupOld, the same cluster numbers are associated to these pairs of nodes in relation relClusterOld,

which are respectively 11 and 12. Nodes without links in relDupOld have a unique cluster number in relClusterOld (nodes 3 and 4).

4.13 Associate

The associate operator is used for mining by association rules. It can be applied to data in both textual and numerical formats, and is designed to deal with usage data. For any data format, the user can choose to return either frequent item sets, association rules, or sequential rules [23]. Although sequential rules is not the most common technique in association rules mining, it might be quite useful for usage log mining, in which the action of the user can be observed in an evolutionary way.

Apart from the rule method (one of the three referred above) and the values of support and confidence, the associate operator has three options regarding the representation of items and transactions: *Word*, *Attribute* and *Log*.

For option Word, each document (i.e., a tuple of a node relation textual attribute) is a transaction, and each distinctive word is an item. Duplicate words are removed. This approach is a text mining modeling for association rules [15]. Option Attribute is applied to data viewed as relational tables, where an attribute name-value pair is an item [23]. Every transaction contains the same number of items, that is the number of attributes.

Option Log is tipically applied to a node relation representing an usage Web data, although it can also be applied to other relations with similar data format. A tipical usage data set of search systems is organized with a session identifier, a query text and/or identifier, and clicked document identifier and/or URL. Figure 4 presents an example of such data set, without data for the query, in relation relUsLog.

For option Log, a transaction is a session. Users must choose one of the following possibilities to represent items: each different word in queries in the same session, each different query in the same session, or each clicked document as an item. For the last possibility, relation relUsLog in Figure 4 would have, for instance, documents 1, 2 and 4 in transaction 1, which is referent to the session number 1. It is not difficult to see that items 1 and 2 represent the most frequent itemset.

The output is represented by a labeled graph. A start node represents an item in the left side of the rule implication, and an end node represents an item in the right side, whereas the label is an identifier for the rule. If relation relAgg in Figure 4 was the result of the associate operator, this graph would mean that, for instance, in rule number 1, items 1 and 2 would imply in item 4, and in rule 2, item 3 would imply in item 4.

From this representation, users can observe the output graph in their analysis or use other WIM operators to mamage data before analysing. For example, using operator aggregate, it is possible to recover rules with several items involved, or to discover the number of occurrences of each item in the left or right side of an implication.

4.14 Analyze

The analyze is an operator for link analysis. It is used for measuring the relevance of a node in a graph, according to its connectivity. The input is a link relation and the output is a node relation, where for each node in the input link relation is associated a value of relevance, according to one of the following methods: *pagerank*, *in-degree*, *authority* or *hub*.

The analyze is an operator for link analysis. It is used for measuring the relevance of a node in a graph, according to its connectivity. The input is a link relation and the output is a node relation, where for each node in the input link relation is associated a value of relevance, according to one of the following methods: *pagerank* [31], *in-degree*, *authority* or *hub* [21].

At the bottom of Figure 4, the analyze operator is applied to relation relPruned, returning the attribute 'u.pr' (whose values are estimated just for exemplification), which is materialized into relation relUsDocs. Observe relation sn1PR in Figure 1. This relation has the attribute 'pr', which stands for Pagerank. Despite the absence of the operation in the figure, the analyze operator was previously applied to the link relation that represents this dataset, with option pagerank, in order to generate relation sn1PR with attribute 'pr'.

4.15 Relink

The relink operator adds new links to a link relation, according to one of the following methods: *co-citation* (bibliographic co-citation [34]), *coupling* (bibliographic coupling [20]) or *transitivity* [17]. A new attribute stores the distance among nodes of the relation. Existing links have the label 0, and new links have label 1.

For co-citation and coupling options, it is not obvious in which direction to include a link and between which pair of nodes sharing a co-citation or coupling unit a link may be inserted. Regarding the direction, the user can choose both directions or a single direction. Regarding the number of links, the user can choose to add a link between all the pairs which share a co-citation or coupling unit, or to add links only between adjacent nodes.

For example, consider the link relation relUsGraph in Figure 4, where start and end nodes are represented in two columns. For instance, start node 1 links to end nodes 1, 2 and 4. The application of the relink operator with option co-citation, single direction, and adding links only between adjacent nodes, results in the same links existent in the input graph with label 0, plus the links represented in graph relCocit, with label 1. For instance, for node 1, links are inserted from 1 to 2 and from 2 to 4, which are the adjacent cocited nodes. The example of Figure 4 includes the selection of only nodes with label 1, represented in relation relCocit with the lable omited.

5 Architecture

In this section we present the architecture of the WIM model implementation, which is illustrated in Figure 6. The system consists of six main modules: Compiler, Executor, Indexer, Visualizer, Pre-processor and Web crawler.

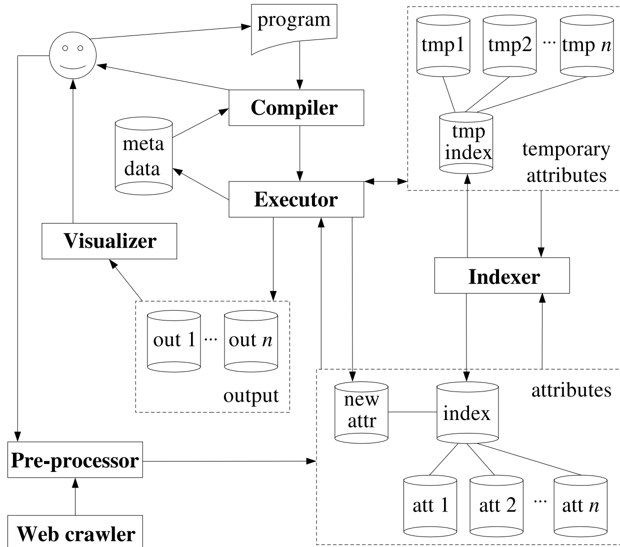

Figure 6: WIM architecture.

The data flow is initiated at the Web crawler and Pre-processor. The *Web crawler* is responsible for collecting the Web data, whereas the *Pre-processor* translate the data into a format that WIM can recognize. As it is shown in the figure, users can provide datasets by other means other than from using the crawler module, which it is required in order to allow for the manipulation of private datasets. After processing a WIM program, the *Visualizer* module presents the output to the user through a friendly interface.

The Web crawler and the Visualizer modules are not native of WIM. There is a number of open source crawlers, and specific tools for visualization of graphs and analysis of statistical results, from which we will add-on to WIM, so that these modules are not the focus in this paper.

The *Indexer* is a module that checks for the need of indexing attributes of relations, receiving an attribute as input and returning the index for that attribute, which is represented in Figure 6 as a single database (named 'index').

Regular and temporary attributes are treated differently by WIM. Temporary attributes are volatiles to the program and most are not stored on disk (exception is for large textual attributes). Regular attributes are materialized as part of existing relations, and can be opened by different programs.

All the metadata is stored in a database. The administrator user needs to set up this database, providing information about each relation and the data to be represented. Examples of metadata are the relation type and size (number of tuples), its list of complementary materialized relations, the number and list of attributes, the type of each attribute, the physical path of the data and its format, among others. The meta database also stores the syntax of the WIM language, which is required in case the user needs to modify the language, like for example in order to include a new operator or an option for an operator.

The main modules of WIM are the Compiler and the Executor, on which we have

concentrated most of our research. The *Compiler* has two important tasks. First, it has to parse the WIM program, according to the data model of Section 3 and the operators of Section 4, and verifying that it is free of syntax and semantic errors. If no syntax errors are found, the compiler recognizes the tokens, such as operators, input and output relations, and specific options for each operator.

The second important task of the compiler is to generate a main function in C. For that, the compiler uses the tokenized input WIM program to select the previously defined function that must be called, for each existing operator. The selection is mainly based on the type of the input relation, on the type of the attribute involved, and on the options requested. Furthermore, in order to generate the main function, variables must be created and memory must be correctly allocated. Control functions, for example, to convert data types, are often needed. Finally, the compiler has the task of compiling the C code, using an external C compiler.

The *Executor* module receives the C program from the WIM compiler. While the program is running, attributes are loaded on demand, and temporary attributes may be created to store temporary data. At the end, the output is presented to the user in the form of files. If required by the user, the result can be materialized as a relation or an attribute of a relation, so that can added as data for future use. This action requires the Executor module to update the metadata.

6 Implementation of WIM

In this section we present the implementation aspects we consider relevant, when materializing the WIM model into the WIM tool prototype.

Web mining applications have two important differences with respect to regular database applications:

1. Although the Web content is highly dynamic, Web data sets are **static**, as a result of crawling a Web snapshot, under a specific domain. It means that deletion and insertion of tuples does not need to be considered, taking into consideration that it is more efficient to create a new data table for new crawls than inserting tuples with different properties into an existent data table.

2. Most Web mining applications do not deal with various attributes together: only the text or the graph structure is used in many cases, so the use of several attributes together is not frequent.

Taking into account these differences, each attribute of a relation must be stored and managed independently one of each other, so that the WIM storage architecture is column-oriented.

WIM uses only two data types to represent elements: floating points and strings. An exception is for attributes that represent identifiers, which are integers. For other attributes, a character must be treated as a string, and an integer must be treated as a floating point.

This solution is a way to substantially simplify the design and implementation of the WIM tool prototype, without significant loss of time and space performance and without any loss of functionality. We refer to the WIM data types as numerical and textual, instead of floating points and strings, respectively.

Temporary relations may physically exist, may be only a *view* of an existing relation, or may have a set of attributes that really exist and a set that are views. For instance, relation sn1New in Figures 1, generated from relation sn1PR, does not have all the attributes existent in sn1PR physically redundant.

Once the WIM operators syntax is defined, the compiler can be implemented to translate operations and parameters to function calls in C. We have created a set of functions for each distinct operator. Roughly, the strategy for designing the functions depends whether the operator accepts both link and node relations, accepts the manipulation of attributes of different types, and sometimes it depends also on the options, when different options imply the use of a different number of attributes. This organization aims to pass to each function only the attributes and parameters that it needs.

Thus, if an operator accepts only one type of relation and one type of attribute, and its options do not imply in using different sets of attributes, the compiler does not need to choose between functions when that operator is used in a WIM program, because only one option is available. This is the case for most of the operators: analyze, calcGraph, compGraph, cluster, disconnect, relink and search. Notice that specific options are managed beyond the interface between the compiler and the particular implementation of the operator, which means that internally the implementation of each operator can have as many functions as the developer wants, in order to keep the code clean.

Operators aggregate, associate and calculate need more than one function, due to the need of different sets of attributes according to different options used. For instance, option single of the aggregate operator demands only one extra attribute be informed, whereas option grouping demands two or three attributes, still depending on sub-options.

Some operators accepts both node and link relations as input: aggregate, calculate, join, set and select. In this case a different function is called for each option. Especially, the join and set operators also accepts manipulation of different types of attributes: integers (for manipulation of identifiers), numerical and textual attributes. Then, different functions are also required to treat each case separately.

Regarding the manipulation of attributes, each numerical attribute of a relation is entirely opened in the main function and passed to other functions as paramenters. Memory must be allocated and is disalocated as soon as the attribute is no longer in use. This pre-opening action does not happen with textual attributes, because they may be too large to be managed in the main memory, so that textual attributes are loaded on demand by internal functions for each operator implementation.

A very important property of the WIM model is the input/output uniformity among operators, that is, the guarantee that any relation of a given type (either node or link) can be input of any operator defined to have that type of relation as input. This property is also valid for the type of the attributes involved. For example, as the compare operator accepts only node relations and needs a textual attribute, if operator disconnect returns a

node relation that is compatible to other node relation that has a textual attribute (so that this textual attribute is inherited), then this output can be input of the compare operator, in any situation.

In order to guarantee uniformity among operators, WIM is designed with just a few different types of data structures, to be used by main functions automaticaly generated by the compiler. Table 1 presents the WIM data structures that the compiler can use (internal functions can use other structures).

Table 1: WIM data structures known by the WIM compiler. Elements within '{}' are structures, preceded by the corresponding name. '*' means an array.

	identifier	numerical
node relations	int*	tPair: {int id, float val}*
link relations	tGraph: {int start, int num, int end}*	tLabGraph: {int start, int num, {int, float}*}*

According to Table 1, there are different representations for node and for link relations. There are also two different representations concerning the type of the attribute, so that structures in column identifier are used in input or output of operations that do not need other attributes apart from identifiers (for instance the input of the analyze and disconnect operators), and structures in column numerical are used when the identifier must come with another attribute, because the operation manipulates other attributes (for instance the ouput of the analyze and disconnect operators, whose relation includes a new attribute).

As textual attributes are represented only by their identifiers, there is no a special representation for textual attributes. Internal functions for each operator receive only the identifiers to open and read the correponding textual attributes.

As an example about how the WIM tool prototype manages attributes according to the each data structure presented in Table 1, consider for instance the disconnect operator. The following is the declaration of the function that must be called when users request this operator:

```
tPair *Disconnect(tConfig inConfig, tGraph
inGraph, tConfig *outConfig, tPair *outNode,
char *attName);
```

where argument inConfig represents metadata for the input relation, and argument outConfig represents metadata for the output relation, which inherits many properties from inConfig, including information about compatibility among relations.

The disconnect operator receives a link relation and returns a node relation, with a new attribute (recur to Section 4.12 if needed). By definition, this operator does not use

any attribute of the input graph, apart from the start and end nodes, reason for why the input is of type tGraph, which is a data structure type defined in column identifier of Table 1. On the other hand, the output is not simply a list of identifiers, given that this operator demands the inclusion of a new attribute in the output node relation, reason for why structure tPair is needed, which is the type for representing numerical attributes for node relations. Argument attName passes the name of the new attribute to be included in the output relation.

Coming back to the issue of uniformity among operators, the question is how can uniformity be guarateed if there are different data strucutures for the same type of relation? Simply by using conversion functions. For example, if there is an attribute represented by structure tLabGraph and there is the need of calling an operator that is defined to receive structure tGraph, the conversion from tLabGraph to tGraph consists of just excluding values in tLabGraph. Notice that if this excluded attribute needs to be used later in another operation, the original tLabGraph is kept.

The opposite situation is when a tGraph structure is returned by an operation and needs to be input of an operator that receives a tLabGraph structure. If this operation deals with attributes of a graph and the user wants to use it, there must exist an attribute for that input link relation. Then, the compiler calls a specific pre-defined function for reading that attribute and associate a value from this attribute to each link in tGraph, converting a tGraph structure into a tLabGraph structure. The same policy is used for node relations.

7 Use Case Analysis

In this section we present five use cases of the WIM model. They are didactic examples of use of the operators, showing how they can be combined in order to return a useful data for the user. illustrations of relations and operators are presented in Figures 10, 2, 3, 4 and 5, for each case in this section.

7.1 Studying the Evolution of Textual Content

The first use case is a study about the evolution of the textual content in the Web. Consider two snapshots of Web documents under a certain domain. We refer to the older collection as old and to the more recent as new. We want to find pieces of documents of the old snapshot that appear in documents of the new snapshot, so that the genealogical tree of the Web is studied. A genealogical tree on the Web is a representation for parents, which are sources of copy, and children, which have copied content, in different snapshots of a given Web subset.

The main problems we have to address regarding the genealogical tree study are: to identify duplicates innerly in a snapshot, in order to avoid the association of multiple parents to a child; to compare parts of every document of the old snapshot against every document of the new snapshot; to filter old-new pairs with the same URL, which means

```
// Clustering duplicates for both old and new collections:
relDupOld = Compare(relOld, sparse, total, at.text);
relClusterOld = Disconnect(relDupOld, conn., newat.clus);
relDupNew = Compare(relNew, sparse, total, at.text);
relClusterNew = Disconnect(relDupNew, conn., newat.clus);
// Comparing the collections:
relSearch = Search(relClusterOld, relClusterNew, shingles, 20%, at.text, at.text);
// Eliminating children with the same URL of parents:
relSearchUrl = CompGraph(relSearch, total, at.url, at.url, newat.sim);
relSeDifUrl = Select(relSearchUrl, value, ==, 0, at.sim);
// Translating start and end nodes into instance nodes:
relStart = Set(relClusterOld, relSeDifUrl, int., at.id, at.start);
relStartInst = Aggregate(relStart, grouping, count, at.clus);
relEnd = Set(relClusterNew, relSeDifUrl, int., at.id, at.end);
relEndInst = Aggregate(relEnd, grouping, count, at.clus);
// Merging instance nodes with the similarity graph:
relGenEnd = Set(relSeDifUrl, relEndInst, int., at.end, at.id);
relGenSt = Set(relGenEnd, relStartInst, int., at.start, at.id);
// Selecting only one parent per child:
relGenFinal = Aggregate(relGenSt, grouping, count, at.end);
```

Figure 7: WIM program to study the textual evolution of the Web.

the documents are the same in both snapshots; and to select one parent when a child has various, which may occur for near-duplicate parents.

Figure 7 presents the WIM program for the Web evolution study application, which is briefly explained below. Notice that most of the relations have a sample represented in Figure 5, and each operator applied to those relations are already explained and exemplified in Section 4, so that we will not present details on relations and operators and will focus on the semantic aspect.

Two materialized relations are input: an older Web collection, relOld, and a more recent collection, relNew. Initially the compare operator is used to identify duplicates in documents within each collection. As a graph is returned, the operator disconnect is used to associate a cluster identifier to each node of the graph. As the output is compatible to relation relOld for the older dataset, all the attributes of relOld are inherited by relation relClusterOld (names of attributes are prefixed by 'at.', and names of new attributes are prefixed by 'newat.'). Note that Figure 5 does not show the steps to generate relation relClusterNew, because they are the same as to generate relClusterOld.

Next, the search operator is used to compare documents between collections relClusterOld and relClusterNew. The result is relSearch, a link relation whose start nodes are compatible to relation relClusterOld and end nodes are compatible to relation relClusterNew. The next step is to filter pairs with the same URL, which are not parent-child

pairs. The compGraph operator is used to compare the URL of each linked pair. For the example in Figure 5, only pair 4–24 has the same URL. Links whose nodes have different URLs are then selected.

At the right side of Figure 5, the set operator is used to return the intersection of elements in relClusterNew and end nodes in relSearchDifUrl. Additionally to the aggregate operator applied to attribute 'clus' of relEnd, these operations are used to identify instances of documents, that is, to identify duplicates within a dataset. The same steps exist to identify duplicates in the start node of relation relSearchDifUrl, returning relation relStartInst, which is omitted in Figure 5.

Then the filtered nodes remained in relEndInst are used to effectively filter the graph relSearchDifUrl, first for the end nodes and later for the start nodes, which is represented in Figure 5 with a set* operation, because the other input relation of the set operator is relStartInst, which is not represented in the figure. The last operation, which is also not represented, is the aggregation of end nodes, in order to associate only one parent to each child.

7.1.1 Comparison with an Ad-hoc Solution

For this application, before implementing a solution using WIM, we implemented an ad-hoc solution in C, which was the natural way once WIM did not exist. In our implementation all the programs had together approximately 2,500 lines of code, and took 1.5 month to be implemented by an advanced programmer. The pseudo-code is presented in reference [3] and the results of the real programs were used to make an extensive analysis on the Web genealogical tree, using five Chilean Web datasets, crawled in different periods of time.

In contrast, the WIM program presented in Figure 7, which took one day to be written, is shorter than the **pseudo-code** of the ad-hoc solution. The program has been run for the same data sets that the ad-hoc solution was run, and very similar results have been found (the slight difference is due to minor difference in internal functions of the two solutions).

Regarding efficiency, we do not compare the WIM program with the ad-hoc program because the code of the ad-hoc program is not optimized, and in fact takes much longer than the WIM program. Our comparison supposes that the ad-hoc program must have at least all the comparison steps of the WIM program in Figure 7, which are the operators compare (applied to both old and new datasets) and search.

Notice that WIM programs are multi-threaded, as the operators follow a dataflow approach. Hence we can readily process the output as it is generated, which exploits the parallelism of multi-core processors. This functionality allowed that the three comparison operators ran in parallel. Then we also suppose that the ad-hoc implementation would run the comparison steps in parallel. The result is that the whole WIM program took 9.02 hours to run, from which only 6.7 minutes were spent with the other operators that are not comparison operations.

```
// Materializing graph relation linking sessions and clicks:
Materialize(relUsLog, relation, link, at.ses.id, newrel.relUsGraph, newat.start);
Materialize(relUsLog, attribute, at.clic.doc, rel.relUsGraph, newat.end);
// Associating urls of the usage log:
relClickedDocs = Aggregate(relUsLog, group., count, at.url);
Materialize(relClickedDocs, relation, node, at.url, newrel.relUsDocs, newat.url);
// Associating the pagerank to clicked documents:
relDocsPr = Join(relUsDocs, relLargeCol, at.url, at.url, at.pr, 0.0);
Materialize(relDocsPr, attribute, at.pr, relUsDocs, newat.pr);
// Manipulating the usage graph to calculate the usage PR:
relFull = Relink(relUsGraph, cocit., single, order, newat.w);
relCocit = Select(relFull, value, at.w, !=, 0);
relAgg = Aggreg.(relCocit, group., at.link, count, newat.qtt);
relPruned = Select(relAgg, value, at.qtt, ¿, 20);
relUsPagerank = Analyze(relPruned, pagerank, newat.u.pr);
Materialize(relUsPagerank, attribute, at.u.pr, relUsDocs, newat.u.pr);
```

Figure 8: WIM program to study the usage Pagerank.

7.2 Studying a Usage Pagerank

The second use case is a proposal of usage pagerank, i.e., a document relevance weight based on a click graph. According to the assumption that the clicks flow within a user session most of the times indicates that the lately clicked documents are more relevant for that query, we propose to compose a graph with the order of clicks within a section and study the pagerank for this usage graph. As an example of such graph, suppose that for a session the user clicked in a page A, then in B and then in C. The graph is composed with the addition of a link from A to B and another from B to C. A single graph is produced as a result of processing all sessions.

Figure 8 presents the WIM program for the usage pagerank application. Notice that most of the relations have a sample represented in Figure 4. The objective of this program is to return a new relation where each tuple represents a document existent in the query log, so the relation must have the following attributes: the URL of the document; the real pagerank, which is not calculated but joined from another dataset; and the new usage pagerank. With this output dataset we can propose a re-ranking function to re-arrange documents in results of queries, based on the comparison of the real and the usage pagerank.

The program has two datasets as input: a Yahoo! query log containing 22 million clicks, represented by relation relUsLog in Figures 8 ('Us' stands for 'Usage') and 4, and a Web dataset from United Kingdom, with 77 million entries, from where the pagerank data is taken, represented by relation relLargeCol.

The program starts by materializing the click data as a graph, so that each session identifier in relation relUsLog becomes a start node in relationrelUsGraph, and every clicked document identifier becomes an end node. The aggregate operator is applied to relUsLog,

so that relation relClickedDocs contains only different URLs, without replication. The 'url' is materialized into relation relUsDocs, to which an automatic identifier is associated to each URL. In the next steps the pagerank and the usage pagerank will also be materialized into relation relUsDocs.

The pagerank for a subset of the documents in relUsDocs is taken from relLargeCol. The join operator is used to compare the 'url' attributes of both the two relations and associate a pagerank for the found URLs. In the example of Figure 4, URL 'w.c' is not found in relLargeCol, then a default value is associated to the corresponding entry in relUsDocs. Attribute 'pr' of relation relDocsPr is then materialized into relation relUsDocs (which is omitted in Figure 4).

The graph relUsGraph, previously materialized from relUsLog, is finally used to calculate the usage pagerank. The relink operator is used to insert new links to the relUsGraph, which is returned in relation relFull. Observe that the relink options used allow the introduction of new links only between adjacent end nodes for each start node. By selecting only links with label 1, relation relCocit, which contains only the new links introduced by the relink operator, is the usage graph we want to calculate the new usage pagerank.

Then the links are aggregated, so that links in relation relAgg are labeled with the number of clicks performed by every user from a given document to other. In this example we select links according to a given threshold, and the returned graph is used to calculate the pagerank for its nodes, which are later materialized as attribute 'u.pr' into relation relUsDocs.

7.3 Studying the Linkage Evolution

The first use case is a comparative study of linkage evolution between new pages with new content and new pages with old content (duplicated content). The hypothesis is that the in-degree, i.e., the number of links that point to a page, of duplicate or near-duplicate Web pages evolves less than the in-degree of Web pages with new content. The intuition behind this hypothesis is that duplicated pages are, most of the times, not returned by search engines, and consequently they are not found and not linked.

The result of such a research may be important for Web crawler designers. On the one hand, the duplication problem is solved in order to avoid indexing duplicated content [25], but not to avoid fetching a page and discovering that it is a duplicate. These tasks consume time and space of search engines computational resources. On the other hand, the Web is very dynamic, and a large number of new pages are published every day. Due to resource limitations, search engines are not able to index all the new pages found every day, and many URLs are placed in the frontier, where the URL is known but the content has not been fetched yet [14]. The frontier problem is to select the frontier pages which should be crawled and indexed. What search engines do in order to deal with this problem is choosing the frontier pages with a good connectivity, which can be represented by its pagerank or other page importance measure [13, 4].

If the hypothesis above is proved, then the page importance will be a good heuristic not only for the frontier problem, but also for avoiding crawling duplicates. Furthermore,

Consider three datasets sn0, sn1 and sn2, which represent snapshots of a domain of the Web at three different times, t_0, t_1 and t_2;

First study the pagerank at a time t_1:

1 Recover the new pages sn1N from sn1, which are the pages whose URL did not exist in t_0 for sn0

2 Compare page content of sn1N against sn0, so as to obtain sn1NDup, that is the list of near-duplicates in sn1N with respect to sn0;

3 Recover the list of documents in sn1N with new content, that is sn1NUni = sn1N − sn1NDup;

4 Calculate the pagerank (or other page importance measure) for pages in sn1NDup and pages in sn1NUni;

Now study the pagerank at a time t_2:

5 Recover sn2Dup, which are pages in sn2 whose URL was represented in sn1NDup;

6 Recover sn2Uni, which are pages in sn2 whose URL was represented in sn1NUni;

7 Calculate the pagerank of pages in sn2Dup and pages in sn2Uni;

8 Compare the pagerank for the different sets.

Figure 9: Algorithm to study the linkage evolution for duplicated and new-content pages.

it will be possible to study different page importance measures, in order to decrease the number of crawled duplicates and improve the resources availability.

Figure 9 presents an algorithm to start studying the problem presented above. The algorithm has two stages: first the pagerank is calculated for time t_1, and second it is calculated for time t_2, comparing both duplicated and new-content pages.

Figure 10 presents the WIM program that implements the algorithm presented in Figure 9. Figure 1 presents an illustration of the relations and operations. Initially the pagerank of pages in sn1 is calculated, using the link relation sn1Link, which is the compatible link relation for sn1. This operation is not illustrated in Figure 1. The search operator is used to identify pages in sn1PR that previously existed in sn0. These pages are represented as end nodes of the sn1URL output link relation. Operator set returns the difference between the snapshot at time t_1 (sn1PR, containing Pagerank), and sn1URL. Relation sn1New represents the new documents, with new URLs, in collection sn1.

The next step is to identify pages in sn1New that are duplicates from some page in sn0, using shingles [7] as comparison method. The result is sn1New2, a link relation whose end node represents pages in sn1New that are duplicates from sn0. The aggregate

```
// First study the pagerank at a time t1:
sn1PR = Analyze(sn1Link, pagerank);
sn1URL = Search(sn0, sn1PR, total, at.url, at.url);
sn1New = Set(sn1PR, sn1URL, dif., at.key, at.end);
sn1New2 = Search(sn0, sn1New, shingle, 100, at.text, at.text);
sn1New3 = Aggregate(sn1New2, group., count, at.end);
sn1NDup = Set(sn1New, sn1New3, int., at.key, at.end);
sn1NUni = Set(sn1New, sn1New3, dif., at.key, at.end);
avg1NDup = Aggregate(sn1NDup, single, avg., at.avg);
avg1NUni = Aggregate(sn1NUni, single, avg., at.agv);
// Now study the pagerank at a time t2:
sn2PR = Analyze(sn2Link, pagerank);
sn2DupURL = Search(avg1NDup, sn2PR, total, at.url, at.url);
sn2UniURL = Search(avg1NUni, sn2PR, total, at.url, at.url);
sn2Dup = Set(sn2PR, sn2DupURL, int., at.key, at.end);
sn2Uni = Set(sn2PR, sn2UniURL, int., at.key, at.end);
avg2Dup = Aggregate(sn2Dup, single, avg., at.avg2);
avg2Uni = Aggregate(sn2Uni, single, avg., at.avg2);
```

Figure 10: WIM Program to study the average pagerank evolution for duplicated and new-content pages.

operator is used to return sn1New3, from which duplicated entries at the end attribute are excluded. The intersection between sn1New and the end node of sn1New3 is avg1NDup, an instance from sn1New representing only duplicated documents. The difference between these relations is avg1NUni, which represents new-content documents in sn1. The aggregate operator is used to return a relation containing the average pagerank for both sets of pages, for time t_1.

Notice that in Figure 1, various relations do not have all their attributes shown. For instance, relation sn1NUni also has the attributes 'text' and 'url', inherited from sn1New. Some operations are prefixed by a character '*', which means that another attribute is input but cannot be shown in the figure, but are explicit in the WIM program in Figure 10.

At the second part, the pagerank of pages in collection sn2 is calculated, generating sn2PR. This operation is not shown in Figure 1. Pages in avg1NDup and avg1NUni, from collection sn1, whose URLs remain existing in collection sn2, are returned, respectively in relations sn2DupURL and sn2UniURL. They are link relations whose end nodes represent pages in sn2. The set operator is used to return documents from sn2PR that are, respectively, duplicated and new-content pages in collection sn1, that remain existing in sn2. Finally the average pagerank is calculated for duplicated and new-content pages, for time t_2.

7.3.1 Preliminary Results

We ran the WIM program presented in Figure 10 for a Chilean Web collection set. Each collection was crawled by the Chilean search engine TodoCL[3]. These collections are very representative, given that the complete list of the Chilean Web primary domains were made available and used to start the crawling. Collection sn0 was crawled in August 2003, collection sn1 was crawled in January 2004, and collection sn2 was crawled in February 2005. They have 3.11, 3.13 and 3.14 million pages, respectively. Details about this dataset can be found in reference [2], where an extensive analysis on the Web content evolution is presented.

Table 2 presents the average pagerank for duplicated and new-content pages, for the Chilean collections presented above. Note that the average pagerank for duplicated pages decreases as time goes on, whereas the average pagerank for new-content pages evolves positively. The results are evidence that the hypothesis presented at the beginning of this section is relevant.

7.4 Manipulating Search Engine Usage Data

Queries submitted to search engines define three main user intent categories, according to the taxonomy of Web search proposed by Broder in [8]. They are: i) navigational, which the users search for a specific reference, possibly already known to him, and once he finds it, he goes to that place and abandons the query session; ii) informational, which the users

[3] http://www.todocl.cl, January 2008.

Table 2: Average pagerank for duplicated and new-content pages, for the Chilean collections 2004 and 2005.

	75% similarity		100% similarity	
collections	dup.	new-cont.	dup.	new-cont.
sn1 (2004)	0.697	0.580	0.714	0.579
sn2 (2005)	0.641	0.740	0.646	0.733
difference (%)	-8.0	27.6	-9.5	26.6

are looking for information about a certain topic and before finding a satisfactory result, they often visits a number of Web pages; and iii) transactional, which the user wants to do some interaction, such as downloading, purchasing, making a bank transfer, or registering to a drawing.

In general, works on user query profiling try to use statistics extracted from query logs to classify queries in one of the categories presented above [22, 33]. However, if we look carefully to some queries we note that it is not possible to put those queries into only one class. For example, if two different users search for “Michael Jackson”, it is possible that one person looks for the Michael Jackson official home page, and the other person looks for information related to Michael Jackson, like news, blogs, images or fan club.

By classifying a query into only one category, the search engines are modeling the need of the majority of the users and those ones with intent deviating from the majority, likely, may not be satisfied. So, the point is that different users have different intents, even for the same query, and the existing models of query intent classification are not considering that.

We have implemented a WIM program to manipulate data from query logs, in order to identify a series of properties for each distinct query that occurs in the log. Taking into account these properties, we have defined functions to identify a large number of queries with both navigational and informational characteristics. Furthermore, we also propose to study *commercial* queries instead of transactional queries, given that search systems cannot take advantage of identifying a transactional query in order to improve the ranking. On the other hand, studying commercial queries can lead to increase the incoming of commercial search engines.

Figure 11 presents the WIM program to manipulate the query log, in order to return the properties to study the search engine user intent. For each distinct query, we need to calculate properties like: the number of sessions that the query appears, the total number of clicks, the number of clicks on sponsored links, the number of sessions with only one click, the number of sessions with only one click on the most clicked document, the number of sessions with more than one click, and the standard deviation of clicks on documents.

We omitted the passage of the program that manipulate queries with sessions with multiple clicks, just to avoid repetition of similar operations. Examples of relations and operations for this application are shown in Figure 2. We run the WIM for a Yahoo! query

log containing 22 million queries. This paper does not extend to the analysis of the results, in order to keep the focus on the WIM.

```
// Materialize new relation with distinct queries and number of clicks:
queries = Aggregate(usageLog, grouping, count, at.query, newat.numClick);
Materialize(queries, relation, at.query, newrel.queryData, newat.query);
Materialize(queries, attribute, at.queryId, rel.queryData, newat.queryId);
Materialize(queries, attribute, at.numClick, rel.queryData, newat.numClick);
// Count the number of clicks on sponsored links:
spon = Select(usageLog, value, at.sponsor, ==, 1);
countSpon = Aggregate(spon, grouping, count, at.sessionId, newat.numS);
numSpons = Aggregate(countSpon, grouping, sum, at.queryId, at.num, newat.numSp);
numSponsJo = Join(queryData, numSpons, at.queryId, at.queryId, at.numSp, 0);
// Count the number of sessions:
click = Select(usageLog, value, at.sponsor, ==, 0);
ses = Aggregate(click, grouping, count, at.queryId, at.sessionId);
numSes = Aggregate(ses, grouping, count, at.queryId, newat.numSes);
numSesJo = Join(queryData, numSes, at.queryId, at.queryId, at.numSes, 0);
// Count the number of sessions with only one click:
countClick = Aggregate(click, grouping, count, at.sessionId, newat.numC);
one = Select(countClick, value, at.numC, ==, 1);
oneClick = Aggregate(one, grouping, count, at.queryId, newat.numOne);
oneClickJo = Join(queryData, oneClick, at.queryId, at.queryId, at.numOne, 0);
// Count the number of sessions with only one click on the most clicked document:
most = Aggregate(one, grouping, count, at.queryId, at.urlId, newat.numMost);
mostOne = Aggregate(most, grouping, max, at.queryId, at.mostOne);
mostOneJo = Join(queryData, mostOne, at.queryId, at.queryId, at.mostOne, 0);
```

Figure 11: WIM Program to manipulate usage data.

The program starts with the aggregate operator used to return the list of queries and the total number of clicks on each query. A relation containing the distinct queries is materialized, named queryData. This action is required if the user needs to access this relation in another program. Attributes queryId and numClick (which is the new attribute returned by the aggregate operator) are also materialized. The next part is the sponsorship data, that is not represented in Figure 2. Relation 'spon' contains only clicks on sponsored links. The aggregate operator is first used to count the number of clicks on sponsors in each session, and later to sum the number of clicks in each session, for each query. The join operator is required to associate each entry in relation numSpons to the correct entry in the new relation queryData.

The next part counts the number of sessions for each query. In the beginning, only clicks on non-sponsored links are selected, returning relation click, which is used in other

```
tfidf1 = Search(queryList, dataSet, tfidf, 120, at.text, at.text, newat.tf);
tfidf2 = Calc.(tfidf1, cons., norm., at.tf2, newat.tf2);
tfidf3 = Select(tfidf2, top, at.tf2, 30);
normPR1 = Calc.(tfidf2, cons., norm., at.pr, newat.pr2);
normPR2 = Calc.(normPR1, pair, sum, at.tf2, at.pr2, newat.tfpr);
normPR3 = Select(normPR2, top, at.tfpr, 30);
tfidfPR = Set(tfidf3, normPR3, union, at.link);
```

Figure 12: WIM Program to compose a pool of documents for relevance assessment.

parts of this program. Operator aggregate is used to count the number of clicks within the same sessionId, for each query that occurs within a session. Observing relation click in Figure 2, we see that only for sessionId 6, the query is the same within the session. Then, in relation 'ses', only one entry is kept for sessionId 6. The aggregate operator is used again to group the queries, and attribute numSes ('n.ses' in Figure 2), which contains the number of sessions in which each query appears, is joined to relation queryData.

In the next part, the aggregate operator is used to count the number of clicks for each session. Relation countClick would be used to process data for multiple clicks, in order to obtain data for calculating the informational characteristic of a query. In this example, countClick is used to count the number of clicks on sessions with only one click. Tuples representing sessions with only one click are selected and returned in relation 'one', and relation oneClick is returned with the number of sessions with only one click, for each query, which is later joined into relation queryData.

Relations with the same queryId and urlId are grouped in relation 'most', where attribute mostOne ('m.one'in Figure 2) contains the number of clicks for each different queryId and urlId pair. Only the greatest values for each queryId are returned in relation mostOne, and are later joined into relation queryData.

7.5 Composing a Pool of Documents for Relevance Assessment

Our third WIM program use case example consists of simply composing a pool of documents returned by different retrieval methods, for a set of queries. As we intend to put our Chilean collections referred in Section 7.3 and usage log available for research, we also want to provide a relevance assessment, so that this data set may be useful in researches on learning to rank. The first part is to select the documents to be judged as relevant, for each selected query.

Figure 12 presents the WIM program for the task of selecting the documents. An illustration of the relations and operations is found in Figure 3. In this program, the results of queries are merged according to two different methods: simple TF-IDF and TF-IDF with pagerank. In the real program we also used BM-25 with and without pagerank, in order to expand the number of documents to be judged.

Initially the search operator is used to retrieve 120 documents for each query, according

to the TF-IDF similarity measure, comparing the text of the query and the text of the document. Figure 3 does not present the attribute text for relation dataSet, which is not important to be illustrated, and only five documents are returned for each query. Relation tfidf1 is a link relation, where attribute 'tf' stores the similarity between the query and the most similar documents. The calculate operator is used to normalize the similarity value, because later these values will be summed to the pagerank, and the values in these attributes may vary too much.

The select operator is used to filter the 30 most similar documents (in Figure 3 only two are filtered), so that the end node of relation tfidf3 contains the list of documents retrieved for the TF-IDF method, for each query. As the end attribute of relation tfidf2 is compatible to relation dataSet, the attribute 'pr' from dataSet can be used as attribute of the links. Then, relation normPR1 contains a new attribute to store the normalized pagerank for each document represented by an end node. The calculate operator is used to sum the normalized pagerank in attribute 'pr2' with the normalized similarity values returned by TF-IDF. Although this attribute is not illustrated in relation normPR1, it exists, because it is inherited from relation tfidf2.

Relation normPR3 contains only the most relevant documents for the method TF-IDF with pagerank. Finally, the results of both different methods used in this application are merged into relation tfidfPR.

8 Concluding Remarks

In this paper we have presented WIM, a model for fast Web mining prototyping. WIM has demonstrated to have some important properties. The operators are *uniformly* designed, which means that any output relation of a given type can be input of any operator that is defined to input a relation of that type. WIM is *scalable* enough to manage several tens of million of tuples in relations, which is the case of the shingles-based comparison used for the Web evolution study, and of the query logs and complementary datasets used for the usage pagerank study.

If the operators are uniform among themselves and the conceptual model is consistent, then the model is easily *extensible*. For instance, for the usage pagerank application presented in Section 7.2, we have effectively implemented an extension for the analyze operator. Instead of selecting links that represent just a few clicks, before the call to the analyze operator, we created a pagerank-like function that is able to include the number of clicks, which is the label of the graph, as a variable of the function. Thus, we simulated an external Web miner contributing to the extension of the model. If respecting some implementation rules, changes of one user can be put available for all WIM users.

Still regarding extensibility, we propose as future work to create a set of operators to deal with machine learning approaches, so that supervised learning tasks can be performed through WIM. With the WIM kernel developed, the task of including a new operator like that is reduced to the implementation job, as the design task now is simple.

WIM has demonstrated to be very *effective* for all the applications we have run so far.

For instance, WIM has successfully been applied to a problem that we first implemented an ad-hoc solution, as shown in Section 7.1. Finally, regarding *efficiency*, as presented in Section 7.1.1, the drawback in using WIM is irrelevant.

For future work, the first step is to make available the WIM tool prototype for general use, so that users interested in new features can contribute with the project. The aggregation of an existent Web crawler is also desirable, to allow users to create their our Web datasets, ready to be used in the WIM tool. Aiming to exploit the parallelism of multi-core processors, we intend to do a map-reduce version based in Hadoop. The aggregation of an existent Web crawler is also desirable, to allow users to create their our Web datasets, ready to be used in the WIM tool.

Acknowledgements

We would like to thank Mariano Consens, Bettina Berendt, Juliana Freire, Jesus Bisbal, and Luciano Barbosa for the valuable detailed revisions of draft versions of this paper. This work was partially funded by Spanish Education Ministry grant TIN2006-15536-C02-01 (R. Baeza-Yates, J. Bisbal, and A. Pereira) and by Brazilian GERINDO Project–grant MCT-/CNPq/CT-INFO 552.087/02-5 (N. Ziviani and A. Pereira), CNPq Grant 30.5237/02-0 (N. Ziviani) and CAPES grant 0694-06-1-PDEE (A. Pereira).

References

[1] G. O. Arocena and A. O. Mendelzon. WebOQL: Restructuring documents, databases, and Webs. In *Fourteenth International Conference on Data Engineering ICDE'98*, pages 24–33, Washington, DC, USA, 1998. IEEE Computer Society.

[2] R. Baeza-Yates, A. Pereira, and N. Ziviani. Understanding content reuse on the web: Static and dynamic analyses. In O. Nasraoui, M. Spiliopoulou, J. Srivastava, B. Mobasher, and B. Masand, editors, *Advances in Web Mining and Web Usage Analysis*, volume 4811 of *LNCS/LNAI*, pages 227–246. Springer, 2007.

[3] R. Baeza-Yates, A. Pereira, and N. Ziviani. Genealogical trees on the Web: a search engine user perspective. In *17th International World Wide Web Conference*, Beijing, China. To appear, 2008. http://www.dcc.ufmg.br/~alvaro/bpz08.pdf.

[4] T. Bennouas and F. Montgolfier. Random web crawls. In *World Wide Web Conference (WWW'07)*, pages 451–460. ACM Press, 2007.

[5] S. S. Bhowmick, S. K. Madria, W.-K. Ng, and E.-P. Lim. Web warehousing: Design and issues. In *ER Workshops*, pages 93–104, 1998.

[6] S. S. Bhowmick, W.-K. Ng, S. K. Madria, and M. K. Mohania. Constraint-free join processing on hyperlinked Web data. In *4th International Conference on Data Ware-*

housing and Knowledge Discovery (DaWaK'2000), pages 255–264. Springer-Verlag, 2002.

[7] A. Broder, S. Glassman, M. Manasse, and G. Zweig. Syntactic clustering of the Web. In *Sixth International World Wide Web Conference (WWW'97)*, pages 391–404, 1997.

[8] A.Z. Broder. A taxonomy of web search. *SIGIR Forum*, 36(2):3–10, 2002.

[9] Soumen Chakrabarti. *Mining the Web: Discovering Knowledge from Hypertext Data*. Morgan Kauffman, 2002.

[10] Junghoo Cho, Hector Garcia-Molina, Taher Haveliwala, Wang Lam, Andreas Paepcke, Sriram Raghavan, and Gary Wesley. Stanford webbase components and applications. *ACM Transactions on Internet Technology*, 6(2):153–186, 2006.

[11] E.T. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[12] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press/McGraw-Hill, San Francisco, CA, 1990.

[13] A. Dasgupta, A. Ghosh, R. Kumar, C. Olston, S. Pandey, and A. Tomkins. The discoverability of the web. In *World Wide Web Conference (WWW'07)*, pages 421–430. ACM Press, 2007.

[14] N. Eiron, K. McCurley, and J. Tomlin. Ranking the web frontier. In *World Wide Web Conference (WWW'04)*, pages 309–318, New York, USA, May 2004. ACM Press.

[15] R Feldman and J. Sanger. *The Text Mining Handbook: Advanced Approaches in Analyzing Unstructured Data*. Cambridge University Press, December 2006.

[16] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for a Web-site management system. *SIGMOD Record*, 26(3):4–11, 1997.

[17] G. H. Gonnet and R. Baeza-Yates. *Handbook of algorithms and data structures: in Pascal and C*. Addison-Wesley Longman Publishing Co., Boston, MA, second edition, 1991.

[18] J. Han and M. Kamber. *Data mining: comcepts and techniques*. Morgan Kaufmann, United States of America, second edition, March 2006.

[19] W. Jicheng, H. Yuan, W. Gangshan, and Z. Fuyan. Web mining: knowledge discovery on the web. In *IEEE SMC'99 Conference*, 1999.

[20] M. M. Kessler. Bibliographic coupling between scientific papers. American Documentation, 1963.

[21] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

[22] Uichin Lee, Zhenyu Liu, and Junghoo Cho. Automatic identification of user goals in web search. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 391–400, New York, NY, USA, 2005. ACM Press.

[23] Bing Liu. *Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data*. Springer, January 2007.

[24] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *5th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297. University of California Press, 1967.

[25] G. S. Manku, A. Jain, and A. D. Sarma. Detecting near-duplicates for web crawling. In *World Wide Web Conference (WWW'07)*, pages 141–150. ACM Press, 2007.

[26] A. O. Mendelzon, G. A. Mihaila, and T. Milo. Querying the world wide Web. *Int. J. on Digital Libraries*, 1(1):54–67, 1997.

[27] DB2 Intelligent Miner. http://www-306.ibm.com/software/data/iminer/, January 2008.

[28] Microsoft SQL Server 2005 Data Mining. http://www.microsoft.com/sql/ technologies/dm/default.mspx, January 2008.

[29] Oracle Data Mining. http://www.oracle.com/technology/products/ bi/odm/index.html, January 2008.

[30] W.-K. Ng, E.-P. Lim, C.-T. Huang, S. Bhowmick, and F.-Q. Qin. Web warehousing: An algebra for Web information. In *Advances in Digital Libraries Conference (ADL'98)*, pages 228–237. IEEE Computer Society, 1998.

[31] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the Web. Technical Report CA 93106, Stanford Digital Library Technologies Project, Stanford, Santa Barbara, January 1998.

[32] S. Raghavan and H. Garcia-Molina. Complex queries over Web repositories. In *Very Large Data Bases (VLDB'03)*, pages 33–44, Berlin, Germany, September 2003.

[33] Daniel E. Rose and Danny Levinson. Understanding user goals in web search. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 13–19, New York, NY, USA, 2004. ACM Press.

[34] H. G. Small. Co-citation in the scientific literature: A new measure of relationship between two documents. *Journal of the American Society for Information Science*, 24:265–269, 1973.

[35] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, San Francisco, USA, second edition, 2005.

[36] M. Yannakakis. Perspectives on database theory. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 224–246. IEEE Computer Society Press, 1995.