A Model for Fast Web Mining Prototyping

Álvaro Pereira[*]
Dept. of Computer Science
Federal 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
Federal Univ. of Minas Gerais
Belo Horizonte, Brazil
nivio@dcc.ufmg.br

Jesús Bisbal
Dept. of ICT
Universitat Pompeu Fabra
Barcelona, Spain
jesus.bisbal@upf.edu

ABSTRACT

Web mining is a computation intensive task, even after the mining tool itself has been developed. Most mining software are developed ad-hoc and usually are not scalable nor reused for other mining tasks. The objective of this paper is to present a model for fast Web mining prototyping, referred to as WIM – Web Information Mining. 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 facilitates the inherently iterative mining process. We present the WIM conceptual model, its associated algebra with a set of operators, and the WIM tool software architecture, which implements the WIM model. We also illustrate how the model can be applied to real Web data mining tasks. The experimentation of WIM in real use cases has shown to significantly facilitate Web mining prototyping.

Categories and Subject Descriptors

H.2.8 [**Information Systems**]: Data Mining

General Terms

Design, Management

Keywords

Web data mining, Web mining applications, model, prototyping

[*]This work was done when at Yahoo! Research Barcelona as a long-term Ph.D. intern.

1. INTRODUCTION

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. 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.

The objective of this paper is to present a model that provides a simple high-level language for accessing and manipulating Web data. The model, referred to as WIM – Web Information Mining –, was designed with the objective of processing data mining tasks for Web mining applications.

WIM is composed of a model and a tool that implements the model. The WIM model includes a data model, a data flow model, and an algebra containing a set of operators. The WIM tool is currently implemented as a prototype, developed to be freely available for Web miners. We also present the WIM software architecture and the main implementation issues of the tool. The WIM language is presented together with use cases, which show how WIM has been used to solve real problems in Web data mining.

Generally, Web mining applications use a combination of the following types of Web data [12, 5]: Web content, such as the text and URL of documents; Web structure, in the form of hyperlinks or implicit links; and Web usage data, in the form of logs, with navigation or application specific data.

Web data mining (or simply Web mining) is an overloaded terminology used to different purposes. We refer to Web mining as the application of data mining techniques to Web data. The WIM algebra was designed observing this definition. For instance, WIM was not conceived to intensive data processing uniquely, like Hadoop [10] and MapReduce [8], which are frequently classified as data mining frameworks due to their capability to find pieces of data among very large datasets, rather than using specific data mining techniques.

There is a large number of data mining techniques that have been applied to Web data [11, 12]. We have included a set of the most common techniques for Web mining in the WIM tool. Through a complementary set of data manipulation operators, WIM allows these data mining techniques to be used sequentially for the same application, integrating the data.

WIM is designed to easily allow the addition of new operations to implement new data mining techniques. Thus, as soon as the first version of the WIM tool is put available, several users will be able to collaborate with the development of new techniques, to be integrated into the WIM tool, gradually improving its coverage of the Web mining world.

We present two real use cases to illustrate how fast the implementation of a Web mining task solution might be when using WIM. The first application concerns a study on the evolution of content on the Web, whose solution we have implemented both ad-hoc [3] and through WIM. The second application concerns the manipulation of search engine usage logs in order to implement a document usage-based relevance weight, with the objective of improving ranking for queries.

2. RELATED WORK

There are two research lines that have some similarity to WIM. The first represents data mining frameworks. Although they are not specially designed for Web data, they may be able to cover some Web mining applications. The second research line represents SQL-like query languages for Web data.

As for data mining frameworks, some commercial SQL databases have add-ons for data mining modules, as Microsoft [14], Oracle [16] and IBM [7]. They have business-driven solutions that are not specific to Web data. Weka [19] is a well known data mining framework, which implements a collection of machine learning algorithms for data mining tasks. The algorithms may either be applied directly to a dataset or called from a Java code. Weka is not designed to Web data and does not provide a specific language to data manipulation, as it is an isolated collection of algorithms.

Influential SQL-like query languages for Web data include WebView[20], Whoweda [15], WebSQL [13], WebOQL [2] and StruQL [9]. They are query languages that model the Web as a kind of relational table, allowing more sophisticated queries and operations than the keyword-based query over the text of the document, as search engines provide. The key differences between WIM and these query languages are: i) they are intended either for on-line queries on the Web or for Web site management; and ii) they are not properly designed for Web mining. Generally they have different data models and concepts, and we decided not to use their models because they would not be efficient and effective for Web mining purpose. Our data model is presented in the next section.

WebBase [18] is an important Web warehouse project that also incorporates a query language, whose main goals are to manage large collections of Web pages and to enable large-scale Web-related research. 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 work. Moreover, the WIM advantages include a special effective design of a data model, a data flow, and an algebra for Web mining. In this sense, the WIM model, implemented through the WIM tool, cannot be classified into an existent class of data access or data mining frameworks.

3. A DATA MODEL FOR WEB MINING

Most Web mining algorithms and software currently available have been developed ad-hoc, with specific data structures in mind. The data structures are used to represent the Web data to be mined. Three types of Web data are usually combined with different degrees for mining purposes, namely 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 required to overcome these limitations.

3.1 A Relational View of Web Data

We now provide basic definitions used to describe the data model that will be presented next. They are taken from the standard relational database literature [1].

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*.

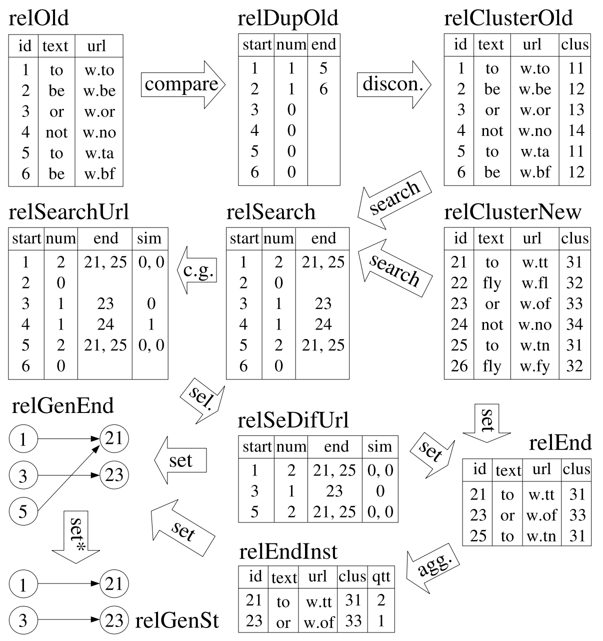
In the WIM model the three types of Web mining data (Web content, Web structure and Web usage data) are represented solely by *relations*, as defined above. Furthermore, only two types of relations will be needed: *node* relations and *link* relations.

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. *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 or clicks 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 *relOld*, *relClusterOld*, *relClusterNew*, *relEnd*, and *relEndInst* are all node relations that represent Web documents in this example. A *node* relation is characterized by its identifying attribute, which identifies the nodes (typically Web documents). It may also include many other attributes such as URL and text of the document, as is the case for these five node relations. Additional attributes can

[1]As described in Section 7, only two domains will be allowed: floating point and string. An exception is for attributes that represent identifiers, which are integers.

be added to relations as needed by the mining task at hand, such as attributes referring to a document's cluster number, as shown by attribute *clus* in some node relations of Figure 1.



relOld

id	text	url
1	to	w.to
2	be	w.be
3	or	w.or
4	not	w.no
5	to	w.ta
6	be	w.bf

relDupOld

start	num	end
1	1	5
2	1	6
3	0	
4	0	
5	0	
6	0	

relClusterOld

id	text	url	clus
1	to	w.to	11
2	be	w.be	12
3	or	w.or	13
4	not	w.no	14
5	to	w.ta	11
6	be	w.bf	12

relSearchUrl

start	num	end	sim
1	2	21, 25	0, 0
2	0		
3	1	23	0
4	1	24	1
5	2	21, 25	0, 0
6	0		

relSearch

start	num	end
1	2	21, 25
2	0	
3	1	23
4	1	24
5	2	21, 25
6	0	

relClusterNew

id	text	url	clus
21	to	w.tt	31
22	fly	w.fl	32
23	or	w.of	33
24	not	w.no	34
25	to	w.tn	31
26	fly	w.fy	32

relSeDifUrl

start	num	end	sim
1	2	21, 25	0, 0
3	1	23	0
5	2	21, 25	0, 0

relEnd

id	text	url	clus
21	to	w.tt	31
23	or	w.of	33
25	to	w.tn	31

relEndInst

id	text	url	clus	qtt
21	to	w.tt	31	2
23	or	w.of	33	1

Figure 1: WIM program to study the Web content reuse and evolution.

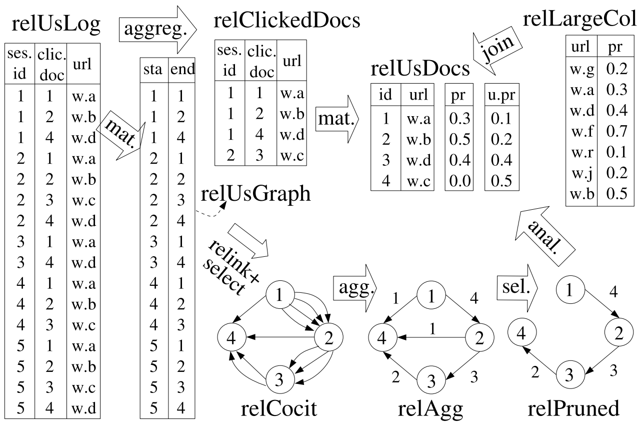
The other relations in Figure 1 are *link* relations, like *relDupOld*, which is illustrated as a table, and *relGenEnd*, which is illustrated graphically in this figure. Link relations must have attributes representing *start* and *end* nodes that are typically represented on a different *node* relation. 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. For some applications, a link relation may have no associated node relation (as we show in the next example in this section), although this situation is not common, since for most applications the existence of links is conditioned to the existence of nodes.

Attribute *end* may contain a list of integers representing the list of nodes to which each start node links (this list can be empty). In addition, link relations may also have other attributes represented as lists of values of the same size as the list of end nodes. Attributes of link relations represent labels of the graph, for example, the anchor text of a link between two Web documents. For instance, the link relation *relSearchUrl* has the attribute *sim*.

The term *compatibility* will be used throughout this paper to refer to a pair of relations associated by a foreign key constraint. In Figure 1, relation *relSearch* is said to be *compatible* with *relClusterOld* as well as *relClusterNew*, since attribute start is a foreign key to *relClusterOld*, and attribute end is foreign key to *relClusterNew*. Notice that node relations frequently have foreign keys to other node relations. For instance, *relClusterOld* has foreign keys to relation *relOld*. In our examples, when the identifier attribute of node relations, or the start or end attributes of link relations, are the same numbers in different relations, then the relations are compatible.

Usage data, such as sessions and clicks, are typically represented as node relations. However, as operations for conversion of types of relations are available, depending on the application, usage data can be seen as a link relation.

Figure 2 is another example of how relations can be used to model real Web data, now including usage data. Relation *relUsLog* is a node relation that represents sessions and clicks on documents (identifiers are omitted in this example). This relation is converted into the link relation *relUsGraph*, where the attribute *ses.id* is converted into the start node and the attribute *clic.doc* is converted into the end node. Thus, *relUsGraph* represents a bipartite graph. Note that there is no compatible relation for the start attribute of this link relation.



relUsLog

ses. id	clic. doc	url
1	1	w.a
1	2	w.b
1	4	w.d
2	1	w.a
2	2	w.b
2	3	w.c
2	4	w.d
3	1	w.a
3	4	w.d
4	1	w.a
4	2	w.b
4	3	w.c
5	1	w.a
5	2	w.b
5	3	w.c
5	4	w.d

sta	end
1	1
1	2
1	4
2	1
2	2
2	3
2	4
3	1
3	4
4	1
4	2
4	3
5	1
5	2
5	3
5	4

relClickedDocs

ses. id	clic. doc	url
1	1	w.a
1	2	w.b
1	4	w.d
2	3	w.c

relUsDocs

id	url	pr	u.pr
1	w.a	0.3	0.1
2	w.b	0.5	0.2
3	w.d	0.4	0.4
4	w.c	0.0	0.5

relLargeCol

url	pr
w.g	0.2
w.a	0.3
w.d	0.4
w.f	0.7
w.r	0.1
w.j	0.2
w.b	0.5

Figure 2: WIM program to study a usage pagerank.

Despite its apparent simplicity, the WIM model suffices to model the most common types of Web data need for mining purposes. Web databases are certainly task and user dependent. As illustrated in Section 8 with specific examples, 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 presented in the following section.

3.2 Data Flow Model Through Operations

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 *dataflow* programming language derived from the functional programming paradigm. It models a program as a directed graph of the data flowing between operation, allowing parallelism of tasks when the manipulated data is independent in different parts of the program.

A *temporary relation* is an output of an operation that can be used anywhere in the current WIM program, and can be materialized for future use in other programs.

As mentioned in the beginning of this section, the concept of compatibility is not limited to pairs of node-link relations. Relations originated from the same physical relation are typically *compatible* among themselves and with respect to the original physical relation, and inherit attributes of other compatible relations. WIM algebra takes advantage 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 has as its own attributes the attributes of any other existent compatible relation.

3.3 Efficiency and Scalability

The WIM data model and the data flow model allow efficiency and scalability that depend only on the software architecture and implementation. The design of the WIM as a dataflow modeling language is a great advantage regarding efficiency. First, dataflow languages allow non-dependent parts of a program running in parallel. For example, consider the following program containing three operations:

$$\begin{aligned}&\text{outputRelation}_1 = \text{operator}_1(\text{inputRelation}_1, \{\text{options}\});\\&\text{outputRelation}_2 = \text{operator}_2(\text{inputRelation}_2, \{\text{options}\});\\&\text{outputRelation}_3 = \text{operator}_3(\text{outputRelation}_1,\\&\qquad\qquad\qquad\qquad\text{outputRelation}_2, \{\text{options}\});\end{aligned}$$

Notice that the first and second operations are independent, as they are applied to different input relations, and the second operation does not use the output relation of the first operation. Then, they can run in parallel, whereas the third operation waits both processes to finish, once it needs the output relations of the first and second operations.

The other important advantage of the WIM design as a dataflow modeling language is that the computational complexity of a WIM program is always equals to the computational complexity of the operation with highest complexity. Considering again the example program above, suppose operations 1, 2 and 3 have computational time complexity $O(n)$, $O(n^2)$ and $O(n\text{log}n)$, respectively. As operations 1 and 2 run in parallel, the time complexity of the whole program is given by:
$T = \text{Max}(O(n), O(n^2)) + O(n\text{log}n) = O(n^2) + O(n\text{log}n) = O(n^2)$.

The conclusion is that the complexity of a WIM program depends on the complexity of the algorithms that implement the operators. In general, the operations with highest complexity are the mining operations. Taking into account that an ad-hoc implementation of the mining algorithm has the same complexity as the WIM mining operation, for a given Web mining problem, then a WIM program has the same complexity as the ad-hoc implementation. This guarantees that the WIM model is designed to be efficient, and state-of-the-art implementations of the WIM operations guarantees that a WIM program is as efficient as an ad-hoc implementation of the same task.

WIM does not extend to the development of a specific file system. The first reason for not extending is simplicity, aiming the fast implementation of an initial version of the tool prototype. The second reason is to allow scalability, where a prototype would use the operating file system, and an industrial-scaled implementation could use a distributed efficient file system like HDFS (Hadoop Distributed File System) [4].

Regarding scalability, we suggest three main levels of scales for both data storage and processing architectures, as shown in Figure 3.

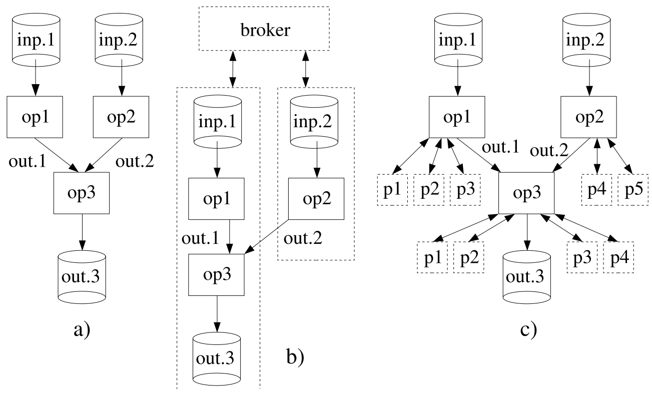


Figure 3: Three architecture levels to support different scales of data storage and processing.

Figure 3(a) refers to the current implementation of our prototype. It is not distributed, and data is stored locally. This architecture is proper for small (around 3 million Web pages) to medium (around 100 million Web pages) datasets, depending on the operation to be processed. Apart from its simplicity, another positive aspect of this architecture is that given operations (e.g., operations 1 and 2) can run in parallel, simultaneously making use of different processors in a multi-processed server.

Figure 3(b) refers to a parallel implementation. It has the advantage of parallel processing, though no gain in storage capacity. It is profitable in case of datasets small to medium sized with high processing. In the figure, dashed boxes mean servers. Observe that one machine is the broker, which has the task of managing metadata, distributing the data, and associating the running of different operations in different machines.

Figure 3(c) refers to distributed processing of each operation. This architecture is the most efficient for large datasets and massive processing, as intensive processing mining tasks can be distributed into as many servers as available or needed. In the figure, The broker is not represented explicitly, but it is the server in which the operations are managed and sent to other servers (processors $p1$ to $p5$). This is our alternative for a future industrial-scaled implementation of WIM, which can run on Hadoop [10].

Observe that the parallelization of the operations does not imply in changing the design of the WIM data model and data flow model, depending only on the implementation, which shows that the WIM model is scalable.

4. AN ALGEBRA FOR WEB MINING

The WIM algebra currently has fifteen operators, which are divided into two categories: data manipulation operators and data mining operators The operators are intentionally presented informally and in summary. Refer to [17] for a detailed description of the operators, with several examples.

4.1 Data Manipulation Operators

Data manipulation deals with retrieval and management of data, often required to prepare the data to be used in another operator or returned in an adequate format.

Select

This operator selects tuples from the input (node or link) relation, according to a condition, such as equal, different, greater than, etc., which is applied to a numeric attribute.

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, only a given number of elements with the highest or lowest values are returned.

Calculate

This operator is used for mathematical and statistical calculations for numeric attributes. Examples of possible calculations are sum, division, rest of division, average.

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.

CalcGraph

Operator CalcGraph applies a calculation operation for every link of a graph, so that the calculation is applied to pairs 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 output is a link relation compatible with the input, with a new attribute to store the result of the calculation.

Aggregate

Operator Aggregate 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 an operation (like sum, average, maximum, count, etc.) is applied to values of a chosen attribute.

Option Single means that an operation is applied to all the elements 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.

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

For example, operator Aggregate with option Grouping is applied to the 'clus' attribute of relation relEnd, in the bottom of Figure 1. 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.

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.

Join

Operator Join is used to add an external attribute in a given relation. This operation is different with respect to the set operation, which returns only the identifier of an input. Operator Join receives two relations, and 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.

In Figure 2, 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.

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 the attributes in use. If the relation to be materialized is compatible to some existing previously materialized relation, an output attribute, which should have been generated in that program, is registered as a new attribute. Otherwise, in case the relation to be materialized is not compatible with other relations, the whole relation may be created, containing at least one attribute. Figure 2 presents two examples of operator Materialize.

4.2 Data Mining Operators

The data mining operators are used to process complete typical Web mining tasks, such as clustering, searching, textual comparison, link analysis, and co-citation analysis.

Search

This operator allows a textual attribute of a node relation to be searched in a textual attribute of another node relation. The typical application for this operator is querying, although it is also important for comparison of texts in general. Several comparison methods have been included, like conjunctive and disjunctive comparisons, TF-IDF, Okapi BM25, exactly the same text, and shingles (text windows).

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

Figure 1 presents an example of operator Search. 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.

Compare

Operator Compare 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 operator Search. The output is a link relation with links between the documents that share a minimal similarity percentage informed by the user.

Operator Compare is applied to the 'text' attribute of relation relOld in Figure 1, using 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.

CompGraph

Allowing the same operations as the Search and Compare operators, operator CompGraph 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 operator CalcGraph. The output is a link relation compatible with the input, with a new attribute to store the similarity of the compared documents.

Operator CompGraph is applied to relation relSearch in Figure 1, for the comparison of attribute 'url' of the compatible relations of start and end nodes, respectively relations relClusterOld and relClusterNew. The output is relation relSearchUrl, where attribute 'sim' 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.

Cluster

Operator Cluster 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. 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.

Disconnect

Operator Disconnect 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. Disconnect is applied to the link relation relDupOld in Figure 1.

Associate

Operator Associate 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 [12].

Apart from the rule method (one of the three referred above) and the values of support and confidence, operator Associate has two options regarding the representation of items and transactions: *Word*, in which a document is a transaction and words are items; and *Log*, in which a session is a transaction and items may be each different word in queries, each different query, or each clicked document.

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 2 was the result of operator Associate, 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.

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 methods like *pagerank*, *in-degree*, *authority*, and *hub*.

Relink

Operator Relink adds new links to a link relation, according to one of the following methods: *co-citation*, *bibliographic coupling*, or *transitivity*. 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.

5. COMPARISON WITH SQL ALGEBRA

SQL (Structured Query Language) is the standard programming language for querying and modifying relational databases [6]. The WIM algebra was not designed observing the SQL algebra, but it was designed after deeply analyzing the most important Web mining tasks, and what kind of data manipulation is needed specially to deal with Web mining applications. In this section we discuss the latter case, with the goal of comparing the WIM data manipulation operators with the SQL algebra, despite their different goals.

Due to the concept of compatibility, WIM does not need to provide an operation to project attributes, which is a fundamental operation in SQL. Another fundamental operation in SQL is selection of tuples, which is used to select tuples according to user-specified criteria on any type of attribute. WIM Select is applied only to numerical attributes, though the selection of textual attributes can be done with the data mining operator Search, and selection of multiple similar documents within a relation, which is not possible in SQL, can be done with operator Compare.

Selection of specific values in Web mining applications is not as common as in relational databases. For example, searching for the name "Catherine Smith" does not mean anything in most Web mining applications, but is important in relational databases, because the tuple for this and other related entries might eventually need to be modified. Then, the WIM algebra is not properly designed to specific selection, or other specific operations. The implication is that operator Search allows a textual attribute of a relation to be searched in a textual attribute of another relation. If the name "Catherine Smith" must be the query, then the user will need to create a relation with only that entry to be used as input of operator Search, which is not as simple as in a SQL operation. For the same reason, WIM does not allow to search using regular expressions (for example, character '%' substitutes any character).

In order to keep the WIM language and implementation simple, the WIM algebra does not provide several conjunctive or disjunctive clauses in a single query. However, users can have the same result by applying a few selection or querying operations in sequence, and using option Union of operator Set for disjunctive selection.

The SQL Order By clause is used to sort results of a SQL query, when showing results to the user. In WIM is the same. An operation for sorting, which is not part of the WIM algebra, is an option for outputting the results. In this paper we do not present details on WIM programs output interface.

Regarding the set operations, SQL allows the union, intersection and difference of query results. A requisite is that the input relations must have the same set of attributes. WIM operator Set does not have this requisite, as a consequence of the WIM data model design. The output relation is compatible to the first input (with exception to option Union), then it does not matter the other attributes that the input relations have. This flexibility is very important to the power of the WIM algebra. Further, WIM Set allows the merging of relations of different types, i.e., link and node relations.

SQL join operations allow multiple attributes of two relations to be joined. As WIM allows only one attribute to be joined, multiple calls to operator Join are required to join multiple attributes. SQL implements different types of join. The main semantic difference among the implementations is how to represent tuples that are not found in both relations. Options include: not representing tuples for missing values, using the value of one of the relations when the value exists, and using a NULL value.

Since for the WIM model it is important to keep compatibility of output relations with respect to input relations, rather than merging all attributes of two relations, WIM operator Join allows only the inclusion of one attribute of the second input relation into the first input relation, so that the output is compatible to the first input. This approach keeps compatibility and solves the need of joining.

A limitation of the WIM join is that only one attribute of each relation can be used for comparison when joining. Although it is a limitation, it produces to a simple and efficient implementation of the operator, and does not seem to expressively limit the applications to which the operator can be applied, due to the nature of Web data. Relational databases tend to have much more sparse data requiring to join multiple attributes than Web databases, as properties of Web objects like documents, link structure and usage data. Web data properties, when available to one entry, are often available to all other entries.

Operations of aggregation are important in SQL and even more important in WIM. The SQL Group By clause allows the same tasks as option Grouping of WIM operator Aggregate.

Regarding database modifications, the WIM data model does not allow the modification of existing attributes of relations. Instead, new attributes can be add or new relations created. SQL allows the update of both numerical and textual attributes, whereas WIM allows operations on numerical attributes, by using operators Calculate and CalcGraph, returning a new attribute as result of a mathematical or statistical operation, but only for numerical attributes. We do not see any advantage in allowing operations on strings, such as adding a prefix, for Web data.

WIM does not allow the insertion of tuples, not required for the designed WIM data model, in which insertion of new Web data is modeled as a new relation in the database.

6. SOFTWARE ARCHITECTURE

In this section we present the software architecture of the WIM tool prototype. Figure 4 presents the six main modules of the system: Compiler, Executor, Indexer, Visualizer, Preprocessor and Web crawler.

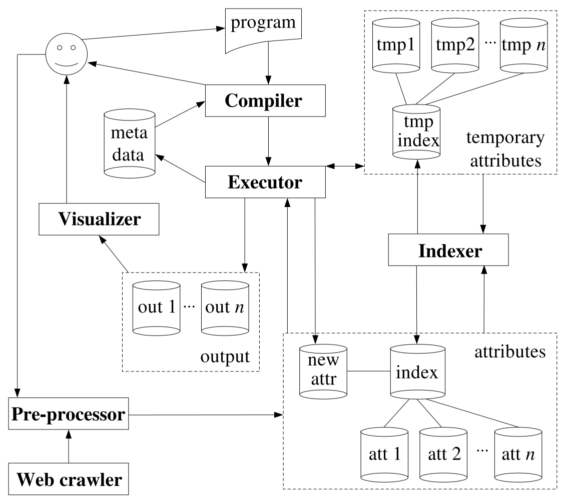


Figure 4: WIM software architecture.

The data flow is initiated at the Web crawler and Preprocessor. The *Web crawler* is responsible for collecting the Web data, whereas the *Pre-processor* translates data into a format that WIM can recognize. Users can provide datasets by means other than from using the crawler module, which is required for the manipulation of private datasets. After processing a WIM program, module *Visualizer* 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.

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 4 as a single database (named 'index'). The Indexer is very important to guarantee efficiency and scalability.

The WIM storage architecture is column-oriented, which means that each attribute of a relation is stored independently of each other. This design decision is explained in Section 7. Regular and temporary attributes are treated differently by WIM. Temporary attributes are volatile to the program and most are not stored on disk (exception is for large textual attributes). Regular attributes are materialized and associated to existing relations, and can be opened by different WIM 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 registers are the relation type and size (number of tuples), its list of compatible 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, in order to include a new operator or an option for an existent operator.

The main modules of WIM are the Compiler and the Executor. The *Compiler* has two important tasks. First, it has to parse the WIM program, taking into account the data model presented in Section 3 and the algebra operators presented in Section 4, and verifying that it is free of syntax errors. Then, the compiler recognizes the tokens, such as operators, input and output relations, attribute names, and specific options for each operator, so that the semantic analyzer can be applied.

The second important task of the compiler is to generate a main function in C. For that, the compiler uses the tokens from the input WIM program to select the previously defined C 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.

Module *Executor* 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, or delivered to module Visualizer.

7. MAIN IMPLEMENTATION ISSUES

This section presents some of the most relevant issues addressed when incarnating the WIM model into the WIM tool. Refer to [17] for further details on the implementation. Web mining applications have two important differences in comparison with regular database applications:

1. Web content is highly dynamic, but Web data sets are **static**, as a result of crawling a Web snapshot. Therefore, deletion and insertion of tuples does not need to be considered, since it is more efficient to create a new data table for new crawls than inserting tuples with different properties into an existing relation.

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.

These differences justify the storage of each attribute of a relation independently of each other, thus the WIM storage architecture is column-oriented.

WIM uses only two data types to represent elements: floating point and string. An exception is for attributes that represent identifiers, which are integers. Characters are treated as strings, and integers as floating points. This solution is a way to substantially simplify the implementation of the WIM tool prototype, without significant loss of time and space performance and without any loss of effectivity. We refer to the WIM data types as numerical and textual, rather than floating point and string, respectively.

The values of an attribute are sorted according to the order in which the identifiers of the relation are represented. Every node relation has an attribute named identifier. Link relations are identified by the start node, which means that a reference to a given start node of the graph appears only once in any physical representation of a link relation. Actually links are physically represented as a list of end nodes for each distinct start node of the graph.

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 *relClusterOld* in Figure 1, generated from the link relation *relDupOld* and compatible with relation *relOld*, does not have all the redundant attributes shown in *relClusterOld*. Thus, in many cases, there is no need to store temporary relations.

As introduced in the previous section, the WIM compiler maps each line of a WIM program into C function calls, as part of module Executor, generating a C file with the main function for that WIM program. It means that there is a set of functions for each operator, referred to as *first level* functions, that are implemented as part of module Executor, but must be known by module Compiler, as part of the interface between these modules. The first level functions are called from the main C function of each WIM program.

Roughly, the strategy for designing the first level functions depends on if the operator accepts link and/or node relations, accepts the manipulation of numerical and/or textual attributes, and sometimes it depends also on the options, when different options imply in using a different number of attributes. This organization aims to pass to each first level function only the attributes and parameters that it needs, saving time and space.

Notice that first level functions include functions not only to implement the operators of the WIM algebra, but also to convert data in different types allowed for WIM, to open attributes, to manage metadata, and to output data in order to be properly shown to the user.

Regarding the manipulation of attributes, numerical attributes of a relation are entirely opened in the main function and passed to other functions as parameters. Memory is released as soon as the attribute is no longer needed. This pre-opening does not happen with textual attributes, due to memory requirements, then textual attributes are loaded on demand.

In order to guarantee uniformity among operators, WIM is designed with just a few different data structures, to be used by main functions automatically generated by the compiler. There are two data structures to represent node attributes (i.e., attributes of node relations) and two other data structures to represent link attributes (i.e., attributes of link rela-

```
// 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 5: WIM program to study the textual evolution of the Web.

tions). Some operators, like Select, return a subset of tuples from the input, without any new attribute, thus a simple array of integers represents the output relation, which is a view of the materialized compatible relation. There is no representation for textual attributes, because texts are not opened and loaded in the main function and passed to first level functions.

8. USE CASE ANALYSIS

In this section we present two use cases to which WIM has been effectively applied. Three other cases are presented in reference [17].

8.1 Studying the Evolution of Textual Content

The first use case is a study about the evolution of the textual content in the Web [3]. Considering 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.

Figure 5 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 1, and each operator applied to those relations are already explained and sometimes exemplified in Section 4, so that we do not present details on relations and operators, focusing on the semantic aspect.

Two materialized relations are input: an older Web collection, relOld, and a more recent collection, relNew. Initially operator Compare 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 1 does not show the steps to generate relation relClusterNew, because they are the same as to generate relClusterOld.

Next, operator Search 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. Operator CompGraph is used to compare the URL of each linked pair. For the example in Figure 1, only pair 4–24 has the same URL. Links whose nodes have different URLs are then selected.

At the right side of Figure 1, operator Set is used to return the intersection of elements in relClusterNew and end nodes in relSearchDifUrl. Together to operator Aggregate, which is applied to attribute 'clus' of relEnd, the two 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 1.

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 1 with a set* operation, because the other input relation of operator Set 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, eliminating near-duplicates in the parents set.

8.2 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 link from B to C. A single graph is produced as a result of processing all sessions in this way.

Figure 6 presents the WIM program for the usage pagerank application. Notice that most of the relations have a sample represented in Figure 2. The objective of this program is to return a new relation where each tuple represents a document existent in the query log. The new 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 6 ('Us' stands for 'Usage') and 2, and a Web dataset from United Kingdom, with 77 million entries, from where the pagerank data is taken, represented by re-

```
// 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(relClickedDocs, relLargeCol, at.url, at.url,
        at.pr, 0.0);
Materialize(relDocsPr, attr., at.pr, relClickedDocs, 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, relClickedDocs,
        newat.u.pr);
```

Figure 6: WIM program to study the usage Pagerank.

lation relLargeCol. We omit the explanation on how the operators work for this use case.

9. DISCUSSION AND CONCLUSIONS

In this paper we have presented WIM, a model and tool for fast Web mining prototyping. WIM has some important properties. The current implementation of the WIM tool is *efficient* to manage several tens of million of tuples in relations, as demonstrated by the use cases presented in Section 8.

The WIM model is designed to be *scalable*. Industrial-scaled implementations of the tool will be able to run distributively, as we indicated in Section 3.3. The current version of the WIM tool, which is not parallel, is scalable under the limits of using one WIM server (optimized when the server is multi-processed), as different levels of indexes may be provided for different amount of data to be processed.

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. Thanks to this property and to a consistent conceptual model, WIM is easily *extensible*. For instance, for the usage pagerank application presented in Section 8.2, we have effectively implemented an extension for operator Analyze. Instead of selecting links that represent just a few clicks, before the call to operator Analyze, 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 pagerank calculation function. Thus, we have simulated an external Web miner contributing to the extension of the WIM model.

WIM has shown to be *effective* for a set of real Web mining problems. The operators are specially designed to take advantage of the association of link and node relations, and also to take advantage of the concept of compatibility, allowing inheritance of attributes and properties after an operation. We have compared the WIM data manipulation operators with the SQL algebra, showing the similarities and explaining the situations in which they are not similar, as they have different purposes.

For future work, the first step is to make available the WIM tool for general use, so that users interested in new features can contribute with the project. The aggregation of an existent Web crawler is also important, in order to allow users to create their own Web datasets, ready to be exported to the WIM tool.

We think that the WIM textual programming language syntax is not so intuitive, but it has been important as a proof of concept before implementing a graphical interface, with which the user will be able to graphically choose operators, input relations and attributes, and options, according to what is previously registered in the meta database. This interface upgrade is important not only to the input, but also to view output results. The tendency is to integrate WIM with a workflow management system for data exploration and visualization, like VisTrails (www.vistrails.org).

Acknowledgements

We would like to thank Mariano Consens, Bettina Berendt, Juliana Freire, and Luciano Barbosa for revisions of draft versions of this paper. This work was partially funded by Spanish Education Ministry grant TIN2006-15536-C02-01 (R. Baeza-Yates, A. Pereira, and J. Bisbal) 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). Also by the Ramon y Cajal Programme of the Spanish Education Ministry (J. Bisbal).

10. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] G. O. Arocena and A. O. Mendelzon. WebOQL: Restructuring documents, databases, and Webs. In *14th Intl. Conf. on Data Engineering (ICDE'98)*, pages 24–33, Washington, DC, USA, 1998.

[3] R. Baeza-Yates, A. Pereira, and N. Ziviani. Genealogical trees on the Web: a search engine user perspective. In *17th Intl. World Wide Web Conf.*, pages 367–376, Beijing, China, April 2008.

[4] D. Borthakur. The hadoop distributed file system: Architecture and design, 2007. http://hadoop.apache.org/core/docs/current/hdfs_design.pdf.

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

[6] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley Publishing, seventh edition, 1999.

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

[8] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[9] 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.

[10] Hadoop, July 2008. http://hadoop.apache.org/.

[11] R. Kosala and H. Blockeel. Web mining research: A survey. *SIGKDD Explorations: Newsletter of the Special Interest Group (SIG) on Knowledge Discovery & Data Mining*, 2, 2000.

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

[13] A. O. Mendelzon, G. A. Mihaila, and T. Milo. Querying the World Wide Web. *Intl. Journal on Digital Libraries*, 1(1):54–67, 1997.

[14] Microsoft SQL Server 2005 Data Mining, July 2008. http://www.microsoft.com/sql/technologies/dm.

[15] 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 Conf. (ADL'98)*, pages 228–237, 1998.

[16] Oracle Data Mining, July 2008. http://www.oracle.com/technology/products/bi/odm.

[17] A. Pereira, R. Baeza-Yates, and N. Ziviani. A model for web mining applications – conceptual model, architecture, implementation and use cases. Technical Report 001/2008, Federal Univ. of Minas Gerais, Feb. 2008. http://www.dcc.ufmg.br/˜alvaro/pbz08b.pdf.

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

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

[20] C. A. Wood and T. T. Ow. WEBVIEW: an SQL extension for joining corporate data to data derived from the web. *Commun. of ACM*, 48(9):99–104, 2005.