User-Defined Redundancy in Web Archives

Bibek Paudel
Max Planck Institute for Informatics
Saarbrücken, Germany
bpaudel@mpi-inf.mpg.de

Avishek Anand
Max Planck Institute for Informatics
Saarbrücken, Germany
aanand@mpi-inf.mpg.de

Klaus Berberich
Max Planck Institute for Informatics
Saarbrücken, Germany
kberberi@mpi-inf.mpg.de

ABSTRACT

Web archives are valuable resources. However, they are characterized by a high degree of redundancy. Not only does this redundancy waste computing resources, but it also deteriorates users' experience, since they have to sift through and weed out redundant content. Existing methods focus on identifying near-duplicate documents, assuming a universal notion of redundancy, and can thus not adapt to user-specific requirements such as a preference for more recent or diversely opinionated content.

In this work, we propose an approach that equips users with fine-grained control over what they consider redundant. Users thus specify a binary coverage relation between documents that can factor in documents' contents as well as their meta data. Our approach then determines a minimum-cardinality cover set of non-redundant documents. We describe how this can be done at scale using MapReduce as a platform for distributed data processing. Our prototype implementation has been deployed on a real-world web archive and we report experiences from this case study.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing– Abstracting methods

General Terms

Algorithms, Performance, Human Factors

Keywords

Web archives, redundancy control

1. INTRODUCTION

Web archives (e.g., the Internet Archive [3] or the Internet Memory Foundation [4]) preserve born-digital web contents as part of our cultural heritage. To this end, they periodically crawl the Web, or parts thereof, and save snapshots of URLs including text documents and embedded media such as images. Ideally, one can later reconstruct the state of a resource as of a specific time in the past from such web archives, making them gold mines for media analysts, historians, and other expert users. Typical sizes of web archives are in the billions of snapshots (e.g., more than 150 billion for the Internet Archive). As a consequence of how data is collected, web archives typically exhibit a high degree of redundancy, containing many snapshots of essentially the same content (e.g., if only typos in a document were corrected) or snapshots that subsume each other (e.g., a blog post with user comments appended over time). Not only does this redundancy waste resources including CPU and storage, but it also deteriorates users' experience, since they have to sift through and weed out redundant content.

One approach to deal with redundancy in web archives is to apply existing near-duplicate detection methods [7, 10]. While viable at first glance, this approach turns out insufficient, as we explain with the following use case:

Carla Columna, a journalist, is reviewing news articles published between 2007 and 2012 to research on the development of the financial crisis in Europe. It is important for her to see articles from different media houses to capture different political views. At the same time, she wants to minimize her efforts and ignore articles whose gist is retained in other more recent articles.

Existing methods that merely identify near-duplicate documents are not helpful here for two reasons. First, in our use case, an article should also be ignored, if its content is absorbed by a much longer, more recent article that is no near duplicate. Second, they can not support Carla's requirement that articles from different sources be retained and her preference for more recent articles.

Our Approach. We propose in this work to equip users with fine-grained control over what they consider redundant. To this end, we let users specify a *binary coverage relation* that encodes when one document is considered redundant in the presence of another document. This binary coverage relation can factor in constraints based on the documents' contents as well as their meta data. For the above use case, one can thus specify that a document covers another document only if (a) it retains a large fraction of the other document's content (e.g., captured using a shingle-based measure and a threshold), (b) was published more recently (i.e., has a later timestamp), and (c) stems from the same source (i.e., has the same host). Once the user has specified her idea of redundancy in this way, our approach determines a minimum-cardinality cover set of non-redundant documents, formally modeled as a *graph cover problem.* Given the size of web archives, we have to ensure that the problem can be addressed at scale. To this end, we describe how the graph cover problem can be solved using MapReduce as a platform for distributed data processing. Here, we eagerly exploit constraints based on meta data to construct our graph more efficiently and leverage observations about typical properties of the considered graphs to decompose the cover problem.

Contributions made in this work include:

- a framework that allows users to define their own notion of redundancy, taking into account constraints based on documents' contents and their meta-data,
- an approach to determine for such a user-defined notion of redundancy a small set of documents representative of the collection,
- efficient algorithms to implement the approach on top of modern platforms for distributed data processing and management, specifically, Hadoop and HBase,
- experiences from a case study where our prototype implementation of the approach was deployed on a real-world web archive.

Organization. Section 2 presents our model of user-defined coverage. We discuss how it can be used to control redundancy in Section 3. In Section 4 we describe our prototype implementation and experiences based on a case study on a real-world web-archive dataset. Related work is discussed in Section 5, before we conclude in Section 6.

2. USER-DEFINED COVERAGE

In this section, we introduce how users can specify a binary coverage relation to express their requirements regarding redundancy. Our focus here is on the conceptual model and we do not consider issues of user-interface design.

Document Model. We operate on a document collection $\mathcal{D}$ – either the entire web archive or a subset thereof selected by the user, for instance, based on keywords. Each document $d \in \mathcal{D}$ is a sequence of terms drawn from a vocabulary $\mathcal{V}$. In addition to their *content*, documents have associated *meta-data attributes* that can be scalar values (e.g., a publication timestamp) or strings (e.g., a URL or host name). Table 1 presents some example meta-data attributes.

Meta Data	Type	Example
`url`	string	http://www.nytimes.com
`hostname`	string	nytimes.com
`timestamp`	scalar	1,339,571,754
`mime-type`	string	text/html
`title`	string	The New York Times

Table 1: Meta-data attributes

Users specify a binary coverage relation $Cov \subseteq \mathcal{D} \times \mathcal{D}$ as a conjunction of content-based constraints and constraints based on meta data. Intuitively, if document d_j covers document d_i (for short: $Cov(d_i,\, d_j)$), the user considers d_i redundant as long as d_j is present, or put differently, it is safe to keep the *covering document* d_j and discard the *covered document* d_i. To ease explanations in the following sections, we assume that the coverage relation is *reflexive* (i.e., $\forall\, d \in \mathcal{D} \ : \ Cov(d,\, d)$), which can always be enforced. We now discuss our two types of constraints in more detail.

Content-Based Constraints operate on a shingle-based representation of documents' contents. For a fixed value of k (e.g., set as $k = 5$), we let $sh(d)$ denote the set of all k-shingles [7] as consecutive sequences of k terms from the document's content. Users can then specify a content-based constraint by combining any of the following three set-similarity measures

- Containment ($C(d_i,\, d_j) = \frac{|sh(d_i) \cap sh(d_j)|}{|sh(d_i)|}$)
- Jaccard ($J(d_i,\, d_j) = \frac{|sh(d_i) \cap sh(d_j)|}{|sh(d_i) \cup sh(d_j)|}$)
- Dice ($D(d_i,\, d_j) = \frac{2 \cdot |sh(d_i) \cap sh(d_j)|}{|sh(d_i)| + |sh(d_j)|}$)

with a threshold $0 < \tau \leq 1$. Choosing $C(d_i,\, d_j) \geq 0.7$, as a concrete example, ensures that on average seven out of ten shingles found in d_i are retained.

Constraints based on Meta Data are assembled from meta-data attributes of two documents d_i and d_j, operators for comparison ($=,\neq,<,>,\geq,\ \leq$) and arithmetic ($+,-,*,/$), as well as scalar or string constants. One can thus, for instance, enforce that the two documents d_i and d_j have the same mime type (i.e., $\texttt{mime-type}(d_i) = \texttt{mime-type}(d_j)$) or that they must have been published within a week's span (i.e., $\texttt{timestamp}(d_i) \geq \texttt{timestamp}(d_j) - 604,800,000$)

Putting things together, we can now define a binary coverage relation to capture the journalist's requirements from our use case in the introduction as

$$\begin{aligned} Cov(d_i,\, d_j) \quad &= \quad C(sh(d_i),\, sh(d_j)) \geq 0.7 \\ &\wedge \quad \texttt{timestamp}(d_i) \leq \texttt{timestamp}(d_j) \\ &\wedge \quad \texttt{hostname}(d_i) = \texttt{hostname}(d_j); . \end{aligned}$$

According to this definition, document d_j covers document d_i only if it originates from the same host, is more recent, and retains at least 70% of the shingles from d_i.

3. REDUNDANCY CONTROL

Having established the idea of a binary coverage relation, we now describe how it can be used to control redundancy.

3.1 Problem Definition

We are given, as an input from the user, a document collection $\mathcal{D}$ and a coverage relation Cov. Our objective, to minimize the user's efforts, is to determine a *cover set* $\mathcal{C} \subseteq \mathcal{D}$ having minimum cardinality yet covering all documents from $\mathcal{D}$. Formally, this can be stated as the following optimization problem

$$\underset{\mathcal{C} \subseteq \mathcal{D}}{\operatorname{argmin}} \mid \mathcal{C} \mid \ \text{ s.t.}$$
$$\forall d \in \mathcal{D} \ \exists \ c \in \mathcal{C} : Cov(d, c) \ .$$

Note that the input document collection $\mathcal{D}$ may comprise the entire web archive or a subset thereof selected by the user, for instance, based on keywords or meta-data attributes (e.g., all documents mentioning international monetary fund that were published between 2007 and 2012). The coverage relation Cov is specified using a combination of constraints as described in Section 2.

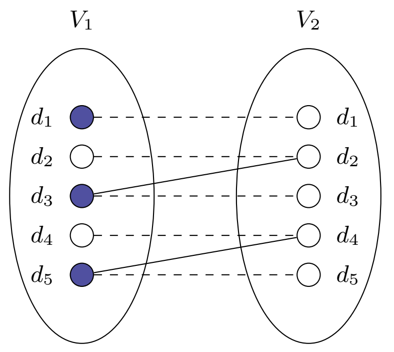

Figure 1: Coverage graph

The above optimization problem can be cast into an instance of Minimum Hitting Set [8]. We can convert a given document collection $\mathcal{D}$ and coverage relation Cov into a bipartite *coverage graph* $G(V_1, V_2, E)$ with vertex sets $V_i = \mathcal{D}$ and edges $E = \{(u, v) \in V_1 \times V_2 \,|\, Cov(u, v)\}$. In this graph, a minimum hitting set (i.e., a minimum-cardinality subset of V_1 such that every vertex in V_2 has an incident edge from a vertex in V_1), corresponds exactly to our cover set. Figure 1 shows an example coverage graph with covering documents as vertices in V_1 on the left, covered documents as vertices in V_2 on the right, and solid vertices that constitute a cover set. Dotted edges in the figure indicate self-coverage due to our reflexiveness assumption.

Coverage graphs become large in practice due to the scale of web archives. Moreover, without additional assumptions about the coverage relation, it is not known upfront which edges exist, so that constructing a coverage graph involves potentially a quadratic number of document comparisons. In the following, we describe our three-stage solution to (i) construct and (ii) partition a coverage graph, before (iii) determining a cover set on it. For each stage, we present a MapReduce solution and, as an alternative for small coverage graphs, sketch a non-distributed solution when sensible.

3.2 Coverage-Graph Construction

The first stage of our solution constructs the bipartite coverage graph. To obtain its edges, we have to materialize the coverage relation from the constraints that the user has specified. Upfront it is not known between which pairs of documents the coverage relationship holds. In the worst case, every document covers every other document, so that a quadratic number of document comparisons is required. This is the case in general for near-duplicate detection techniques and several pruning optimizations (e.g., based on document lengths [17]) exist to reduce the number of comparisons. When constructing the coverage graph, we primarily determine those document pairs that meet the content-based constraints, while using meta-data constraints for pruning. This is done in two steps:

First, as an auxiliary structure, we create a shingle-level inverted index from the document collection $\mathcal{D}$ that associates with every occurring shingle the list of documents that contain it. This step can easily be parallelized using MapReduce. Hence, in the `map()` function we consider each document and emit shingle-document pairs, which are aggregated into per-shingle index lists in the `reduce()` function. Creating a shingle-level inverted index is beneficial, since it allows us, in the second step, to consider only pairs of documents that share at least one shingle.

Second, we materialize the coverage relation. Again, this can be implemented using MapReduce. The `map()`-function considers a per-shingle index list and emits a document-document pair for every pair of documents therein that meet the meta-data constraints and pass the filters applicable for the employed set-similarity measure. For instance, for our use case from the introduction, we only emit a document-document pair if the two documents are from the same host and the second is more recent than the first. In the `reduce()`-function, we count how often every document-document pair occurs, thus determining the intersection cardinality of their corresponding shingle sets, and emit the document-document pair, if it meets the content-based constraints. This way, we push the meta-data constraints deep into the evaluation of the content-based constraints. This is beneficial, since it drastically reduces the number of comparisons and also the amount of data, both in terms of key-value pairs and sheer volume, that a MapReduce implementation transfers over the network.

3.3 Coverage-Graph Partitioning

We found that coverage graphs obtained from real-world web archives are typically very sparse and consist of many connected components. This is natural for relatively high values of the threshold τ in the content-based constraints and can also be an effect of meta-data constraints. As a concrete example, consider that our use case from the introduction leads to a natural clustering of the coverage graph into at least one connected component per host.

Given our problem formulation, connected components constitute independent subproblems. We can thus determine a global cover set as the union of many local cover sets determined on the connected components in isolation. We leverage this property by determining the connected components of the coverage graph upfront. This is beneficial, since it allows us to parallelize the computation of the cover set and we only consider much smaller problem instances.

To determine connected components, we either employ the standard union-find algorithm [11] on a single machine, if the coverage graph is sufficiently small, so that the required bookkeeping can be performed in main memory. Otherwise, for large coverage graphs, we again employ MapReduce. To this end, an adaptation of the parallel breadth-first search described by Lin [13] is used. While the parallel breadth-first search algorithm passes information about distance, we pass information about a vertex representing the connected component. For each adjacency list in the coverage graph, we pick the vertex with the largest identifier as a representative and pass it to the adjacent vertices. In the end, all adjacency lists contain information about the vertex with the largest identifier from the entire component. Identifying members of the components is then a matter of aggregating all vertices containing a common pivot. We found this approach to work well in practice, but note that a more refined approach [18] to computing connected components in MapReduce has been proposed recently.

3.4 Cover-Set Computation

Once the connected components of the coverage graph have been identified, we determine a cover set on each of them in isolation and in parallel. Here, we employ an adaptive approach that considers the size of each connected component. For small instances, consisting of at most ten vertices, we apply a brute-force approach to determine the optimal solution, enumerating all exponentially many solutions. For larger instances, we resort to the well-known greedy algorithm [11], that chooses vertices in the order of their marginal benefit, with its $H(n)$ approximation guarantee.

4. CASE STUDY

To examine the viability of our approach, we implemented a prototype system and deployed it on a real-world web-archive dataset. In this section, as a case study, we report our experiences from this case study.

4.1 Dataset and Setup

We used a real-world web-archive dataset provided by the Internet Memory Foundation [4] as a common benchmark dataset for the LAWA [5] project. It contains regular snapshot of the U.K. parliament website and associated web content from the period 2009 – 2011. The dataset is stored in an HBase [2] table with the URL as row key, content and various meta data in two separate column families as values, and multiple versions per row.

All computations were done on a cluster consisting of ten server-class computers, each equipped with 64 GB of RAM, two Intel Xeon X5650 6-core CPUs, and four internal 2 TB SAS 7,200 rpm hard disks. Debian GNU/Linux 5.0.9 (Lenny) was used as an operating system. Machines in the cluster are connected via 1 GBit Ethernet. Cloudera CDH3u0 is used as a distribution of Hadoop [1] and HBase [2] running on Oracle Java 1.6.0_26. One of the machines acts a master, the other nine machines are configured to run up to ten map tasks and ten reduce tasks in parallel.

4.2 Prototype Implementation

The architecture and data flow of our prototype implementation are illustrated in Figure 2. When computing a cover set for a given document collection, our system goes through the following stages:

1. **Coverage-Graph Construction.** We first construct an inverted index of all shingles in the document collection. This requires one Hadoop job that reads documents from HBase and writes the inverted index to HDFS. Then, a second Hadoop job builds the coverage graph, mapping over each index list and emitting a key-value pair of all document pairs that meet the user-defined meta-data constraints. At the end of this first phase, the coverage graph is available as adjacency lists in HDFS.

2. **Coverage-Graph Partitioning.** In this stage, we first use iterative Hadoop jobs to compute connected components, as described in Section 3.3. After each iteration, a new adjacency list is written out to HDFS which contains information about the neighboring vertex. The document with the highest identifier is chosen as a pivot vertex, which is the representative document for each component. In the end, when all the documents in a component contain information about the pivot document, the computation is terminated.

	# Documents	# Shingles	Documents/URL	
			μ	σ
A	1,667,654	812,331,426	2.24	1.24
B	348,778	240,464,783	2.07	1.24
C	75,299	94,143,006	2.32	0.47

Table 2: Input document-collection statistics

Reduction	A-1	B-1	B-2	C-1
# Documents	50.32%	49.49%	48.59%	58.99%
# Bytes	55.46%	51.09%	50.94%	59.16%

Table 3: Reduction achieved on different scenarios

3. **Cover-Set Computation.** In the final stage, a last Hadoop job maps over the adjacency lists output by the previous stage to bring together all vertices from the same connected component. On its reduce side, a cover set is determined for each connected component using either the brute-force or greedy algorithm, as described in Section 3.4.

Scenarios. We consider three subsets of the above web archive as input document collections:

A All URLs with at least two versions,

B All URLs with at least two versions published in 2010,

C All URLs with at least two versions that are of mime-type `text/html`, contain the term `expense`, and stem from the host `parliament.uk`.

Statistics about our input document collections are given in Table 2. We combine them with two coverage relations:

$$\begin{array}{llcl} (\mathbf{1}) & Cov(d_i,\, d_j) & = & C(sh(d_i),\, sh(d_j)) \geq 0.7 \\ & & \wedge & \texttt{timestamp}(d_i) \leq \texttt{timestamp}(d_j) \end{array}$$

$$\begin{array}{llcl} (\mathbf{2}) & Cov(d_i,\, d_j) & = & C(sh(d_i),\, sh(d_j)) \geq 0.7 \\ & & \wedge & \texttt{timestamp}(d_i) \leq \texttt{timestamp}(d_j) \\ & & \wedge & \texttt{hostname}(d_i) = \texttt{hostname}(d_j) \end{array}$$

resulting in four scenarios **A-1**, **B-1**, **B-2**, and **C-1**.

4.3 Reduction Results

Table 3 shows reduction statistics for these scenarios in terms of documents and storage. On average, all scenarios show a reduction of around 50%, which can be attributed to the average number of versions per URL. We observe that the reduction achieved for scenarios B-1 and B-2 is comparable, since most redundancy occurs anyway within the same host. However, the additional meta-data constraint allows us to compute the cover set more efficiently, as described in Section 3. Similarly, most documents in scenario C-1 talk about financial affairs. These documents are less likely to undergo many changes once they have been created, so that we achieve a higher reduction for this scenario.

4.4 Wall-Clock Times

Since we did not have exclusive access to the cluster, we can only report approximate wall-clock times. For all scenarios, the most time-consuming phase was the coverage graph

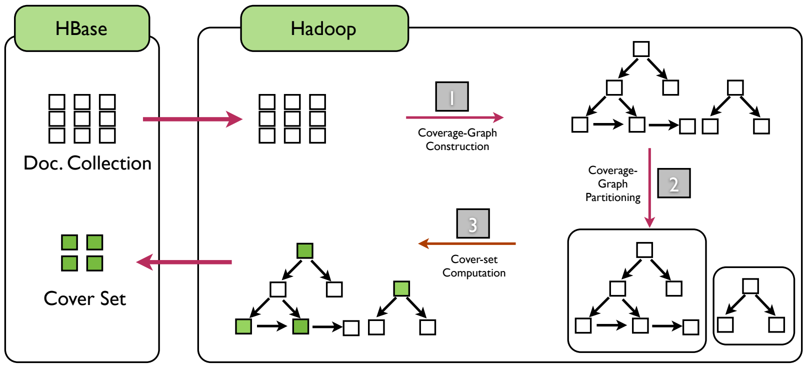

Figure 2: Architecture and data flow of our prototype implementation

construction, taking from about one hour for scenario B-2 to about eight hours for scenario A-1. In all cases, constructing the shingle-level inverted index took less than one hour to complete. We used MapReduce to determine connected components of the constructed coverage graphs. Here, the runtime depends on the diameter of the largest connected component. This stage and the final stage of computing the cover set took less than one hour hour for all scenarios.

5. RELATED WORK

The problem of detecting duplicates or near-duplicates has received attention from different communities. Manber [14], as an early work from the systems community, identify similar files using fingerprints. The notion of shingles was introduced in the seminal work by Broder [7] to detect near-duplicate documents. Since then, researchers have looked intensively into improving the effectiveness [10, 17] and/or efficiency [6, 12] of near-duplicate document detection methods. For an overview of work on the related problem of detecting duplicate records in structured (relational) data, we refer to Naumann and Herschel [15]. In the context of web archives, Gomes et al. [9] investigate how storage of duplicate documents can be avoided already at crawl time. Nørvåg [16], as the work closest to ours, considers and compares approaches to systematically prune document versions from a web archive. However, all of the above approaches assume a one-size-fits-all notion of redundancy, and none of them can easily adapt to user-specific requirements.

6. CONCLUSIONS AND FUTURE WORK

In this work, we have presented a novel approach to deal with redundancy in web archives that gives users fine-grained control over what is considered redundant.

Ongoing & Future Work. Turning the approach presented in this paper into a prototype system that can be employed by web-archive users (e.g., journalists or social scientists) is part of our ongoing work. One challenge here is achieving tolerable response times and providing the user with an estimate when the redundancy-free document collection will be ready. Beyond that, we plan to investigate the following problems: First, how can we support more complex requirements (e.g., retaining at least a minimum number of documents from every host); Second, how can cover sets be maintained dynamically as the document collection changes; Third, how can notions of document quality (e.g., based on popularity) be worked into our approach.

Acknowledgments

Partially supported by the EU within the 7th Framework Programme under contract number 258105 "Longitudinal Analytics of Web Archive data (LAWA)".

7. REFERENCES

[1] Apache Hadoop. `http://hadoop.apache.org`.
[2] Apache HBase. `http://hbase.apache.org`.
[3] Internet Archive. `http://archive.org`.
[4] Internet Memory Foundation. `http://www.internetmemory.org`.
[5] Longitudinal Analytics of Web Archive Data. `http://www.lawa-project.eu`.
[6] R. J. Bayardo et al. Scaling up all pairs similarity search. WWW 2007.
[7] A. Z. Broder. On the resemblance and containment of documents. SEQUENCES 1997.
[8] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman & Co., 1990.
[9] D. Gomes et al. Managing duplicates in a web archive. SAC 2006.
[10] M. R. Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. SIGIR 2006.
[11] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison-Wesley, 2005.
[12] J. Lin. Brute force and indexed approaches to pairwise document similarity comparisons with mapreduce. SIGIR 2009.
[13] J. Lin and C. Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan & Claypool, 2010.
[14] U. Manber. Finding similar files in a large file system. USENIX 1994.
[15] F. Naumann and M. Herschel. *An Introduction to Duplicate Detection*. Morgan & Claypool, 2010.
[16] K. Nørvåg. Granularity reduction in temporal document databases. *Inf. Syst.* 31(2), 2006.
[17] M. Theobald et al. Spotsigs: robust and efficient near duplicate detection in large web collections. SIGIR 2008.
[18] T. Seidl et al. CC-MR - Finding Connected Components in Huge Graphs with MapReduce. ECML PKDD 2012.