# Page Digest for Large-Scale Web Services

Daniel Rocco, David Buttler, Ling Liu
Georgia Institute of Technology
College of Computing
Atlanta, GA 30332, U.S.A.
rockdj|buttler|lingliu@cc.gatech.edu

January 31, 2003

**Abstract**

The rapid growth of the World Wide Web and the Internet has fueled interest in Web services and the Semantic Web, which are quickly becoming important parts of modern electronic commerce systems. An interesting segment of the Web services domain are the facilities for document manipulation including Web search, information monitoring, data extraction, and page comparison. These services are built on common functional components that can preprocess large numbers of Web pages, parsing them into internal storage and processing formats. If a Web service is to operate on the scale of the Web, it must handle this storage and processing efficiently.

In this paper, we introduce Page Digest, a mechanism for efficient storage and processing of Web documents. The Page Digest design encourages a clean separation of the structural elements of Web documents from their content. Its encoding transformation produces many of the advantages of traditional string digest schemes yet remains invertible without introducing significant additional cost or complexity. Our experimental results show that the Page Digest encoding can provide at least an order of magnitude speedup when traversing a Web document as compared to using a standard Document Object Model implementation for data management. Similar gains can be realized when comparing two arbitrary Web documents. To understand the benefits of the Page Digest encoding and its impact on the performance of Web services, we examine a Web service for Internet-scale information monitoring—Sdiff. Sdiff leverages the Page Digest encoding to perform structurally-aware change detection and comparison of arbitrary Web documents. Our experiments show that change detection using Page Digest operates in linear time, offering 75% improvement in execution performance compared with existing systems. In addition, the Page Digest encoding can reduce the tag name redundancy found in Web documents, allowing 30% to 50% reduction in document size for Web documents especially XML documents.

**Keywords:** Web services, document processing, document storage

# 1 Introduction

The promise of Web services offers an exciting new opportunity for harnessing the immense collection of information present on the Internet. To fully utilize the Web as an advanced data repository will require sophisticated data management techniques that will provide the mechanisms for discovering new information and integrating that knowledge into existing stores. Current search engines provide core discovery and classification services, but foundational work is still needed to make large-scale Web services a reality.

One important consideration for large Web services is data storage and processing efficiency. Much of the data on the Internet is contained in HTML documents that are useful for human browsing but incur significant drawbacks from a data management perspective. HTML has no real type information aside from layout instructions, so any data contained in the document is mixed with formatting and layout constructs intended to help browser software render pages on screen. Automated data extraction or comparison of Web pages is expensive and slow.

We introduce a new document encoding scheme to address some of the problems associated with storage and processing of Web documents as a means toward enabling Web service applications to operate efficiently on a large scale. The Page Digest encoding is designed to bring some of the advantages of traditional string digest algorithms to bear on Web documents. A Page Digest of a Web document is more compact than HTML or XML format but preserves the original structural and semantic information contained in the source document. Unlike schemes using general compression algorithms, a Page Digest is not "compressed" from the source nor does it need to be "decompressed" to be used, which minimizes the processing needed to convert a document from its native format. Further, the digest encoding highlights the tree structure of the original document, greatly simplifying automated processing of digests. Finally, the Page Digest is structurally aware and exposes the information contained in a document's tree shape to applications.

The major characteristics of the Page Digest are summarized below.

- *Separate structure and content.* Documents on the Web—such as HTML or XML—can be modeled as ordered trees, which provide a more powerful abstraction of the document than plain text. The Page Digest uses this tree model and explicitly separates the structural elements of the document from its content. This feature allows many useful operations on the document to be performed more efficiently than operating on the plain text.

- *Comparable.* Page Digests can be compared directly to each other. Subsections of the digest can also be compared, which provides the means for semantically richer document comparisons such as

resemblance.

- *Invertible.* Page Digests can be efficiently converted back to the original document. Since the digest encoding is significantly smaller than the original document, it provides a scalable solution for large-scale Web services. The digest is an ideal storage mechanism for a Web document repository.

The rest of the paper proceeds as follows. Section 2 discusses other work in the fields of general document digest, similarity, tree encodings, and change detection. Section 3 presents the Page Digest encoding in detail and examine the process of producing a digest from a standard Web document. Section 4 briefly discusses applications of the Page Digest. Our experiments in Section 5 evaluate the performance advantages realized by applications using Page Digests. We conclude by considering related efforts and discussing future research directions.

## 2 Related Work

**Digest.** There is a large body of existing work on comparison and digest mechanisms for general text and binary strings. One of the earliest applications of these algorithms is in network transmission protocols where they are used to detect and possibly correct transmission errors in blocks of data. A simple scheme is to append a *parity bit* [19] to each block that is chosen to force the number of "1"s in a bit string to be even or odd; the choice of even or odd is agreed upon beforehand. Hamming [11] proposed the notion of the *Hamming distance* between two bit strings and devised a mechanism for using this metric to produce protocols that could correct single bit errors in a block; this technique has been extended to correct burst errors as well [19]. Another family of error detecting algorithms is the cyclic redundancy check (or cyclic redundancy code) [19, 2], which produces a "check word" for a given string. The check word is computed by the sender and appended to the transmitted block. On receiving the block, the receiver repeats the checksum computation and compares the result with the received checksum; if they differ, the block is assumed to be corrupt and the receiver typically requests a retransmission.

Another application of digest algorithms is in the area of information security where they are used to protect passwords, cryptographic keys, and software packages. Perhaps the most popular of the *cryptographic hash functions* is the MD5 [15] algorithm, which is an extension of Professor Rivest's earlier MD4 algorithm [16]. Other examples of similar hash functions include SHA-1 [8] and its variants and the RIPEMD [7] family; SHA-1 is an extension of MD5 while RIPEMD was developed independently. Although the mechanics of these algorithms differ, they have similar design goals.

The three important features for cryptographic applications are pseudo-unique hashing, randomization, and computationally difficult inversion. Suppose we have a hash function $H$. Ideally, if two documents $x$ and $y$ exist such that $x = y$, then $H(x) = H(y)$; conversely, $x \neq y \Rightarrow H(x) \neq H(y)$. However, these hash algorithms produce fixed length results (that is, $|H(x)|$ is constant for any $x$), so guaranteeing a unique hash for any arbitrary input is impossible. Rather, these algorithms are designed for computationally difficult inversion: for a given $H(x)$, finding an $x$ that generates $H(x)$ is computationally difficult by current standards.

The randomization property of these algorithms means that small changes in the input document result in dissimilar hash values. Given $x$ and $x_1$, where $x_1$ is a single-bit modification of $x$, $H(x)$ and $H(x_1)$ will be totally distinct.

In designing the Page Digest encoding, we attempted to capture the utility of string digest mechanisms while leveraging the unique features of Web documents. The Page Digest design diverges from more traditional digest mechanisms with respect to its intended application domain, focusing on efficient processing, comparison, and storage utility rather than cryptographic security or error correction.

**Change Detection and Diff Services** Traditional change detection and difference algorithms work on strings or trees. Past research focused on establishing a set of change operators over the domain of discourse and a cost model for each of the defined operators. Once these parameters were established, the goal was, given two elements from the domain, to construct an algorithm that would produce a *minimum-cost edit script* that described the changes between the two elements with the smallest overall cost of operations. Barnard et al. [1] present a summary of several string-to-string and tree-to-tree correction algorithms. Dennis Shasha et al. [18] summarize their work on various problems related to tree-to-tree correction and pattern matching in trees. Chawathe and Garcia-Molina [6] extend portions of this work with additional semantic operators to make the difference document more meaningful: in addition to insert, delete, and update, they introduce the operators move, copy, and glue to impart greater semantic meaning to the generated diffs. Change detection using these algorithms consists of generating a minimum-cost edit script and testing to see if the script is empty. Algorithms for generating minimum-cost edit scripts for trees are computationally expensive, while string difference algorithms miss many of the nuances of tree-structured data formats.

Others have also recognized the importance of fast change detection that targets tree-structured documents, such as HTML and XML. Khan et al. [13] compute signatures for each node in a tree, allowing fast identification of only those subtrees which have changed. The main cost here is the computation of the signature of each node. The drawback to this approach is that false negatives—documents which change,

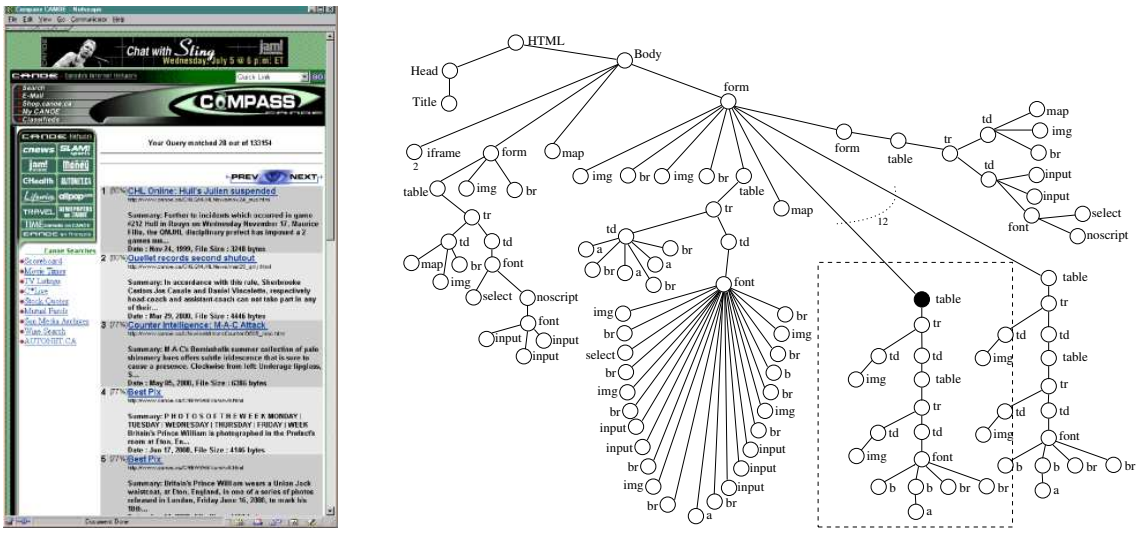



Figure 1: Sample HTML Page and Associated Tree

but the signature of the root node does not change—are possible. Many applications cannot tolerate false negatives, making this approach unsuitable or undesirable for such systems.

We have written a change detection algorithm, Sdiff, that explicitly uses the structural characteristics exposed by the Page Digest encoding. This algorithm deviates from the focus of earlier work by leveraging the unique encoding of the Page Digest to detect semantically meaningful changes. Sdiff explicitly supports many different types of change detection, providing a smooth trade-off between cost and the level of change detail extracted. The supported types of change detection include simple changed versus not changed, efficient markup of exactly which elements have changed between two versions of a document, and computation of a minimal edit script that transforms one version into another. Many applications can use Sdiff's flexibility to achieve better performance and semantic interpretation of data. Using Sdiff, a Web service can compare documents quickly and with more control than text diff algorithms allow, examining particular facets of the document that are interesting and ignoring changes in uninteresting portions. For example, applications such as Web change monitoring services need to detect interesting changes to a specific aspect or subset of a page, but do not need to compute edit scripts. If a Web service requires full edit scripts for proper operation, Sdiff can invoke a powerful tree-diff algorithm that focuses computing resources on only those documents that have been changed.

# 3 Page Digest

## 3.1 Overview

The Page Digest is a straightforward encoding that can be efficiently computed from an ordered tree of a source document, such as an HTML or XML document. Figure 1 shows an example of a rendered HTML page and a visualization of its tag-tree representation.

The Page Digest encoding consists of three components: node count, depth first child enumeration, and content encoding. We selected these particular elements for inclusion in the format based on careful evaluation of Web documents and the characteristics of those documents that are important for many applications. For example, the tree structure of a document can help pinpoint data object locations for object extraction applications.

*Node Count.* The first component of a Page Digest is a count of the number of nodes in the document's tree representation. The main purpose for this count is for fast change detection: if two documents $W_1$ and $W_2$ have different node counts then $W_1 \neq W_2$. Inclusion of this count can eliminate costly difference computations for change detection applications and provides a size estimation for storage management.

*DFS Child Enumeration.* The second component of the digest is an enumeration of the number of children for each node in depth-first order. In a typical Web document encoding such as HTML, the structure of the document is entwined with its content. We hold that separating the content and structure of the document provides opportunities for more efficient processing of the document for many applications. We have already mentioned using structural cues to extract data objects; document structure can also be used to enhance change detection as described in Section 4.

*Content Encoding and Map.* Finally, the Page Digest encodes the content of each node and provides a mapping from a node to its content. Using the content map, we can quickly determine which nodes contain content information. The content encoding preserves the "natural order" of the document as it would be rendered on screen or in print, which can be used to produce text summaries of the document.

## 3.2 Specialized Digest Encodings

In the previous section we examined the general Page Digest encoding that can be applied to many different types of documents. However, if more specific information is available about a particular type of document to be encoded, it is possible to construct a specialized digest format that takes advantage of the nuances of that type. For example, TEX documents and HTML both have additional document features that can be

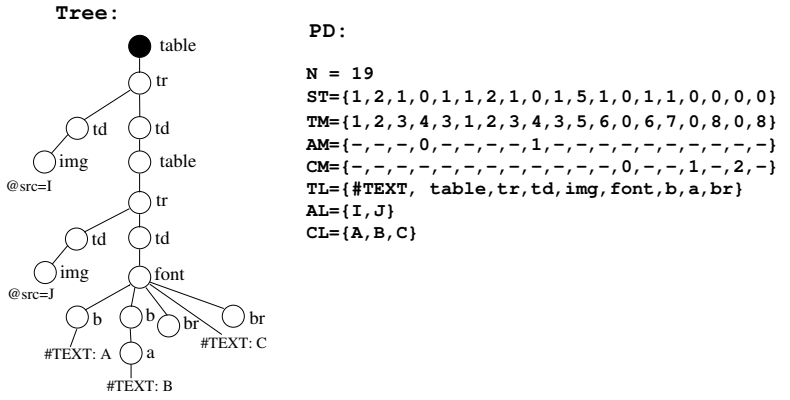



Figure 2: Tree Fragment with Corresponding Page Digest

encoded and used.

We have constructed a specialized digest for HTML pages. In addition to the three components of the general digest format, the HTML encoding includes two additional aspects: tag type information and attributes. All HTML nodes are tagged with a particular type such as "BODY" or "P." While we could encode this information as a form of content, it is advantageous to use a specialized encoding to exploit the characteristics of HTML. Tags tend to be quite repetitive, so using a tag encoding and a map from nodes to their tag allows us to store each unique tag name only once for the entire document, thus saving storage and memory costs. This also provides an efficient mechanism for locating nodes of a particular type.

The specialized Page Digest encoding can also handle XML documents, which are more data-centric than HTML and tend to have longer, more descriptive tag names. Given that tags in XML documents tend to comprise a higher percentage of the overall document, Page Digests for XML documents can present significant saving in terms of storage and memory costs.

### 3.3 Page Digest Example

Consider the example Web document in Figure 1, which shows the tree representation of a page from a Canadian news site [5] with text nodes omitted for clarity of the diagram. This example page is typical of Web pages modeled as trees. It contains two subtrees—rooted at nodes "head" and "body"—that are children of the root HTML node. The head of the document contains the title tag and text of the title, while the body of the page comprises the bulk of the document and contains the content of the page enclosed in several tables, plus a table for the navigation menu and a few images and forms.

The tree fragment in Figure 2 shows one of the table subtrees of Figure 1 with the addition of the node

attributes and text nodes. The node types—or tag names—appear to the right of the node. Attributes appear below the node to which they belong and are denoted with the "@" symbol. All nodes except for text nodes are shown as circles on the graph; text nodes are shown via the label "#TEXT" followed by the text content of the node represented with a single-letter variable. The document fragment represented by this subtree is a table that contains one row with two columns—child "tr" with two "td" children. The first column of the table contains an image while the second column contains another table containing two columns with an image and some text.

To the right of the subtree is its Page Digest representation including the HTML tag and attribute extensions. N shows the number of nodes in the subtree. The representation of the tree structure of the document is contained in the list ST, whose elements are number of children at each node in depth-first traversal order. Notice that the subtree root node "table" has one child: ST[0] therefore has the value '1'. Continuing in depth-first order, "tr" has two children (ST[1] = '2'), the leftmost child "td" has one child (ST[2] = '1'), and the node "img" contains no children (ST[3] = '0'). The encoding continues in a similar fashion along the rest of the subtree in depth-first order.

The array ST has several useful properties. It encodes the structure of the tree and can be used to re-construct the source document or generate the path to a particular node. It also provides a compact representation of the tree. ST can in most cases be easily encoded in only N characters without any bit-packing or other compression techniques. Each node can be identified via the ST array index, which indicates the node's position in the array: the root is "0," the root child node "tr" is "1" and so on. Finally, ST can be used to quickly identify areas of interest such as leaf nodes or large subtree nodes in the tree. The subtree anchored at the "font" node, for example, contains 5 children while every other node has 2 or fewer.

TL, AL, and CL denote a tag list, an attribute list, and a content list respectively. TL is the set of all node types found in the subtree. AL is the set of all the attributes encountered. CL is a set containing any content from #TEXT nodes. TM, AM, and CM denote tag mapping, attribute mapping, and content mapping respectively. They map the node array ST to the type, attribute, and content lists. The nodes are matched to their types, attributes, and contents in the same order they appear in ST. Thus, the root node, 0, is mapped to TM[0] = 1 indicating it has type 1, AM[0] = '-' and CM[0] = '-' signifying that this node has no attributes or content. Looking closer, we observe that TL[TM[0]] = TL[1] = type "table," exactly as we would expect given that the root node 0 is of type "table."

Every node must have an entry in TM since every node has a type. For HTML, the size of TL is typically less than N since tags are usually repeated: in any given page, there is usually more than one node of the

same type, be it a text section, paragraph, table row, or section break. In contrast, having an entry in AM or CM depends entirely upon whether the node has attributes or content. Observe that text content is always found at the leaves of the tree: CM[x] $\neq$ '-' implies that ST[x] $= 0$, where x is the index on the number of nodes in the subtree table ranging from 0 to $(N - 1)$.

### 3.4 Page Digest Conversion Algorithms

Algorithm 1 shows a sketch of the algorithm for converting a source document `W` into its digested form, while Algorithm 2 codes the reverse process. For these algorithms, we assume that a document is *well-formed* [21] with properly nested tags. Due to the space restriction and without loss of generality we only describe the construction of ST array and omit the algorithm detail for constructing tag name (TL, TM), attribute (AL, AM), and content (TC, CM) arrays from these algorithms. An overview of the impact of these facets on the time and space complexity of our algorithms will be reported in Section 3.5.

Algorithm 1 begins by initializing the data its data structures, including a stack that is used to keep track of node ancestry and a `PARENT` array that keeps track of each node's parent index. The main loop of the algorithm reads the characters of the input document `W` until it reaches the end of the input. While it is reading, the algorithm scans the document for start and end tags, which determine the structural information and ancestor context for each node. The end result of Algorithm 1 is the structural array (ST) with node count information.

*Analysis of Algorithm 1.* The algorithm reads the characters of `W` and uses the stack to retain the ancestor hierarchy of the current node. A newly encountered tag indicates a descent down the tree, while an end tag indicates a return up the tree. The time complexity of the algorithm is dominated by line 12, which scans part of the source document `W` at each pass. Each scan is non-overlapping with all other iterations, and each part of `W` is read only once, giving a total time for all iterations of $O(k)$, where $k = |\mathrm{W}|$. The other steps inside the loop are all $O(1)$ operations to within a constant factor, so the remaining steps of the loop require $O(n)$, where $n$ is the number of nodes in `W`; we thus have $O(k + n)$. However, we note that $n$ is at most $\frac{k}{7}$ since each node in `W` occupies a minimum of 7 characters, which yields a final time complexity of $O(k)$ for Algorithm 1.

The space required by the algorithm is dominated by the `PARENT` and `ST` arrays, each of which occupy $O(n)$ characters. In addition, the space required by the stack `S` is bounded by the depth of the tree representation of `W`; in the worst case where W is a linked list, `S` occupies $O(n)$ space at the end of the algorithm.

**Algorithm 1** Page Digest

```
W ← <character array of source document>
S ← stack()
PARENT = []
ST = []
index = 0

/* root has no parent */
PARENT[0] = '-'
push(0, S)

while W ≠ EOF do
  tag = readNextTag(W)
  if isStartTag(tag) then
    if index ≠ 0 then
      PARENT[index] = peek(S)
      ST[PARENT[index]]++
      ST[index] = 0
    end if
    push(index, S)
  else
    pop(S)
  end if
  index++
end while
```

Algorithm 2 shows the process of converting from a Page Digest back to the original document. This process is modularized into two components. The main routine maintains a visited list V to track nodes that have already been seen during the tree traversal. The stack S maintains the ancestor hierarchy and preserves sibling order for the traversal. The CHILDREN array keeps a list of each node's children, which is obtained from the *children* subroutine. Initially, the stack contains only the root node. The main loop of the algorithm proceeds by popping the top node from the stack; this is the current node. If the current node has not yet been visited, three actions occur: the open tag for that node is output, the node is pushed back on the stack and the visited list, and the node's children are pushed onto the stack from right to left so they will be visited in left to right order. If the current node has already been visited when it is first popped off the stack, then the algorithm has finished traversing that node's children so it outputs the node's closing tag.

The *children* subroutine takes a Page Digest and constructs a list of the indices of each node's children. To accomplish this task, it requires a stack CS to maintain the node hierarchy. The subroutine first constructs an array of empty lists to hold the children of each node. Then, starting from the root node, it visits each node in ST keeping the current node's parent as the top element of the stack. Every node is added to its parent's CHILD list. If the current node is an internal node, the next nodes in the depth-first ordered ST array will be the children of the current node, so the subroutine pushes the current node onto the stack. For leaf nodes, the subroutine checks the top node on the stack to see if the current node is its last child; if so,

**Algorithm 2** Page Digest to original document

```
P ← Digest of Page W
ST ← P.ST
V ← {}
S ← stack()
CHILDREN ← children(P)

/* push the root onto the stack */
push(0, S)
while |S| > 0 do
  current ← pop(S)
  if current ∉ V then
    push(current, S)
    V ← current ∪ V
    print('<' + type(current) + '>')
    for c ∈ reverse(CHILDREN[current]) do
      push(c, S)
    end for
  else
    print('</' + type(current) + '>')
  end if
end while

/* subroutine to find children of all nodes */
N ← nodeCount(P)
CS ← stack()
ST ← P.ST
CHILD = []

for index = 0 to (N - 1) do
  CHILD[index] = {}
end for

/* push the root onto the stack */
push(0, CS)
for index = 1 to (N - 1) do
  parent = peek(S)
  add(index, CHILD[parent])
  if ST[index] ≠ 0 then
    push(index, CS)
  else
    CHILD[index] = '-'
    if ST[parent] = |CHILD[parent]| then
      pop(CS)
    end if
  end if
end for
```

the top node is popped off the stack and the routine continues.

*Analysis of Algorithm 2.* We first consider the complexity of the main portion of the algorithm. Initialization of variables—excepting the CHILDREN array, which we will discuss shortly—can be done in constant time. Although a cursory inspection of the body of the algorithm suggests $O(n^2)$ operations due to the nested loops, careful inspection reveals that the inner `for`-loop will visit each non-root node exactly once, thereby executing exactly $n-1$ push operations over the course of all iterations of the outer loop. Additionally, we can ensure that the $reverse$ operation incurs no additional cost by traversing the CHILDREN lists from the end rather than the front. For the visited list check, if V is handled naïvely as a list, the algorithm will execute approximately $2n$ searches of V, a list of average size $\frac{n}{2}$. However, using a hash table for V will yield constant time searches and inserts. The other operations contained in the outer loop execute in constant time. Thus, the complexity of the algorithm arises in part from the cost of pushing all $n-1$ non-root nodes onto the stack in the inner loop. We have an additional cost of $c \cdot 2n$ for the outer loop where $c$ is a constant representing the total cost of operations for the outer loop. This yields a total cost for both loops of $(n-1) + c \cdot 2n$.

Computation of the CHILDREN lists involves several constant time variable initializations plus $n$ empty list creations in the first `for` loop. The body of the algorithm loops $n-1$ times over the nodes of the Page Digest, performing several constant time stack manipulations, comparisons, and list additions. Total cost for the loop is $d \cdot (n-1)$, where $d$ represents the cost of the operations of the loop.

Our final cost for Algorithm 2 is the cost of computing the CHILDREN list plus the cost of the output routine. The total cost is $n + d \cdot (n-1) + (n-1) + c \cdot 2n$, yielding a final time complexity of $O(n)$.

### 3.5 Further Analysis

Algorithms 1 and 2 show Page Digest conversion on simplified documents for clarity of presentation, but converting actual Web documents adds only a few additional processing requirements. The Page Digest shown in Figure 2 captures all the components of the document by encoding the type, attribute, and content lists. These mappings can be built up at the same time that the structure array ST is constructed with little additional computation overhead. The space overhead for the mappings AM, TM, and CM is $3n$. Redundant tag names are eliminated, so each unique tag name in the document is stored once; we will demonstrate the savings afforded by this technique in the experiments. The attribute and content lists consume the same amount of space as in the source document.

We assume in the above algorithms that Web documents are well-formed: proper nesting and closing

tags are assumed to be in place. Although proper nesting is an absolute requirement, it is possible to relax the strict requirement for closing tags in some circumstances. For instance, HTML specifications have historically allowed tags such as "br" that do not have an end tag. In such cases, we can generate an implied closing tag using rules that are specific to each tag type; it is also possible to correct some HTML errors this way.

# 4 Page Digest Applications

Now that we have seen the Page Digest encoding, we consider several application examples that make use of the Page Digest to increase their operating efficiency.

## 4.1 Document Comparison

The Page Digest provides several advantages over standard Web document formats for change detection and diff generation applications. First, the digest separates the structural elements of a Web document from its content, allowing the change detection and diff algorithms to run independently of each other to speed up the change detection step. The Page Digest is designed to make the important structural information of a Web document explicit and store such information in a compact form that can easily be scanned for changes. For example, after the encoding of a Web document is complete, the number of nodes in the document and a list of types are both immediately available and easily comparable. It is also easy to identify leaf nodes and their contents. This separation is not present in many other diff tools including Unix `diff` [10], which must compute the entire diff by scanning all the bytes in the input files to report whether or not a change has occurred.

We have incorporated the Page Digest's design principles into a Web change monitoring application—Sdiff. Sdiff uses the unique characteristics of the Page Digest to perform efficient structurally-aware change detection, change location, and difference generation on Web documents. It's algorithms operate over structural, tag, attribute, and content facets of the Web document. For instance, comparing node counts and tree structures between two document digests can be done with $O(n)$ comparisons, where the number of nodes $n$ is significantly smaller than number of characters of the source documents. This technique is much more efficient than even the best tree difference algorithms, leading to significant efficiency improvements for change detection over document sets with even occasional structural differences. Algorithm 3 shows an example of this structural change detection that might be used as part of a Web change monitoring application. For further details on the other algorithms in the Sdiff suite, see [17].

**Algorithm 3** Structural Change Detection

```
S = Original Page Digest of page at time t_o (loaded from disk)
W = Page at time t_1

let D = PageDigest(W)
if S.size ≠ D.size then                                          5
  /* Digests have different number of nodes */
  return true
else
  let i = 0
  for i < S.size do                                              10
    if S.ST[i] ≠ D.ST[i] then
      /* Digests have different structure at node i */
      return true
    end if
    i++                                                          15
  end for
end if
return false
```

Intuitively, structural change detection first checks if the number of nodes in the two documents is the same; if not, at least one node has been inserted or deleted, so a structural change has occurred. However, if the sizes are the same, the algorithm must still check each node since it is possible for the nodes to have been rearranged. If at each node $i$ in both documents the number of $i$'s children is the same, then no structural change has taken place.

### 4.2 Web Data Extraction

Web data extraction services try to automatically detect and extract objects from Web documents, where "objects" are defined to be the data elements found on the page. Non-object content includes advertisement, navigational, and formatting constructs that do not make up the information conveyed in the document. These data objects can then be used by Web services for tasks such as integrating data from different sites; automation allows such services to extract the data components of a page without site-specific clues or markup conventions. As an example, one application allows a user to enter queries about books and see relevant information compiled from several online book vendors. The benefit of the Page Digest for Web data extraction stems from the digest's compact format and fast access to tag type information that is used to determine object boundaries. An example Web data extraction system is Omini [4].

### 4.3 Document Similarity

Another interesting application in the document comparison domain is similarity estimation between documents. Traditional message digest algorithms provide exactness detection, but it is sometimes useful to

know if two documents are "similar" [3] in some fashion: Web search engines that provide similar pages to a given search result might employ this type of estimate. With the Page Digest, we extend traditional similarity estimate ideas to provide a more semantically rich set of similarity comparisons. Two intuitive comparisons are structure-only and content-only similarity, but it could also be useful to compare the documents at a specific set of nodes: for instance, one might pose the question "What is the similarity between the paragraphs of these two documents?"

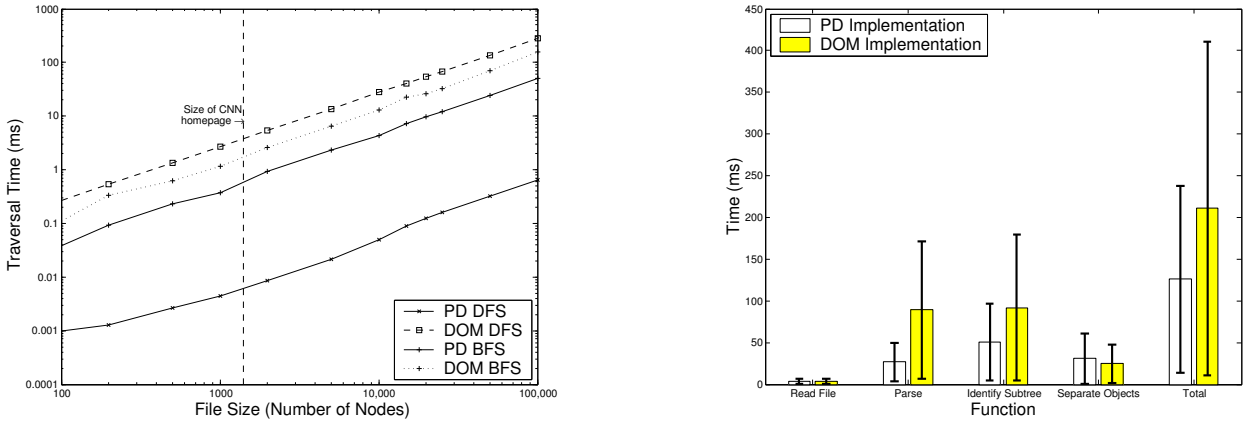
# 5 Experiments

We report two sets of experiments. The first set of experiments is designed to test various characteristics of the Page Digest when performing some common operations over Web documents. The second set of experiments tested the first version of our Sdiff implementation on several data sets with the goal of evaluating Sdiff's performance characteristics relative to other tools. We will first discuss our experimental setup by noting the equipment and language features used to run our experiments. We also discuss the test data including how we obtained the data set and what the test data is intended to model. Finally, we outline our experimental goals and present the results.

## 5.1 Experimental Setup

**Equipment and Test Data.** All measurements were taken on a SunFire 280 with 2 733-MHz processors and 4 GB of RAM running Solaris 8. The software was implemented in Java and run on the Java HotSpot Server virtual machine (build 1.4.0, mixed mode). The algorithms were run ten times on each input; we averaged the results of the ten executions to produce the times shown.

We used two sources of data for our experiments: data gathered from the Web and generated test data. The Web data consists of time-series snapshots of certain sites and pages collected from a distributed set of sites. For the generated data, we built a custom generation tool that can create XML documents containing an arbitrary number of nodes with a user specified tree structure. The data used in these experiments consists of documents of various sizes with three basic tree shapes: bushy, mixed, and deep. We used data with different shapes to model the variety of data found in different application scenarios and to test the sensitivity of tree processing applications to tree shape.




(a) (b)

Figure 3: Page Digest Traversal and Object Extraction

## 5.2 Page Digest Experiments

The suite of experiments outlined here are designed to test various characteristics of the Page Digest when performing some common operations over Web documents. Our experimental goals were to model real Web data, to test the strengths and weaknesses of the Page Digest algorithms, and to gauge performance relative to similar tools.

### Experiment 1: Tree Traversal

Figure 3(a) shows a comparison of the cost of traversing an entire document tree for both the Page Digest and DOM. The Page Digest is able to complete traversals faster than the DOM implementation used in our tests; using the Page Digest for DFS traversals is equivalent to scanning an array of size $n$, which accounts for the Digest's huge performance advantage for depth first search.

### Experiment 2: Object Extraction Performance

Object extraction from Web documents occurs in four phases: read file, parse, identify subtree, and determine object delimiters. This experiment uses the Omini [4] object extraction library, which uses a document tree model for its processing routines. In this experiment, we show the object extraction performance using an off-the-shelf DOM implementation compared with a re-implementation using Page Digest as the processing format.

Of the four phases, we expect the reading time to be constant for both systems since they both use the same IO subsystem. The parsing and subtree identification phases rely on tree traversals to function, which

we expect to be an advantage for the Page Digest encoding. Object delimiter determination operates on nodes of the already-traversed tree, which does not immediately present an advantage to either approach.

Figure 3(b) shows the average and maximum performance times for Omini using Page Digest and DOM. The graph bars show the average performance time for all data samples in the experiment. We have included error bars that show the range of times obtained using our experimental data set: the low end of the bar is the fastest time recorded while the high end is the slowest time. As we expected, the time to read the page was identical between the two implementations. The DOM implementation edged out the Page Digest version for object separation minimum, maximum, and average case performance, while the Page Digest implementation performed markedly better in most cases for parsing and subtree identification.

**Experiment 3: Size Comparison**

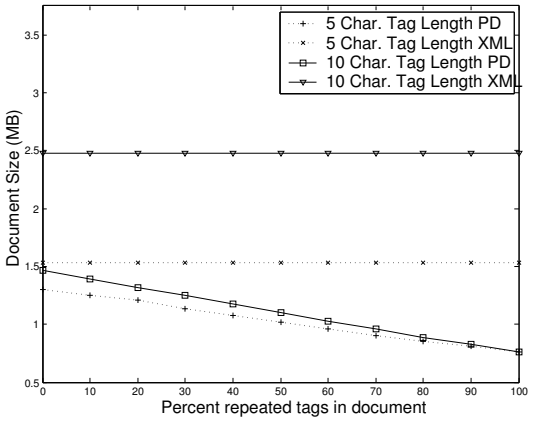
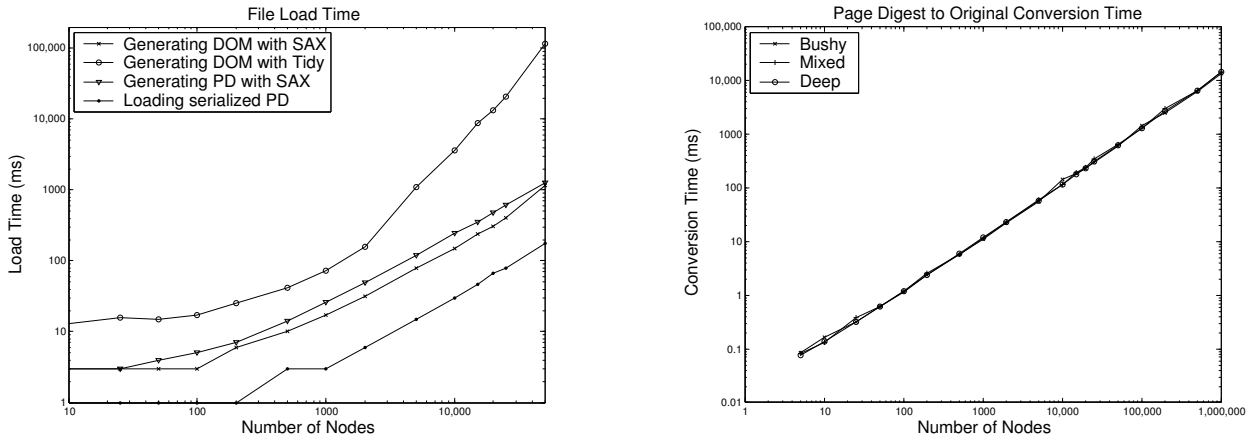



Figure 4: Page Digest Size Comparison

Experiment 3 presents a size comparison between two documents in XML format and their equivalent Page Digests. The Page Digest encoding, while not a traditional compression algorithm, can reduce the size of documents by eliminating redundant storage of tags in the source, present in both HTML and XML. Naturally, the savings afforded by this feature vary with the type of data being digested: content-heavy documents with few tags will benefit the least, while highly tag-repetitive data-centric documents will enjoy the largest reduction in size.

This experiment examines the potential reduction offered by the Page Digest; Figure 4(a) shows the results. All tag sizes in the data were fixed at 5 or 10 characters. The x axis in the graph is the percentage of tag repetitions: the 50% mark, for example, shows the compression when 50% of the document's tags are the same and the other 50% are all unique.




(a) (b)

Figure 5: Page Digest to Web Document Conversion Performance

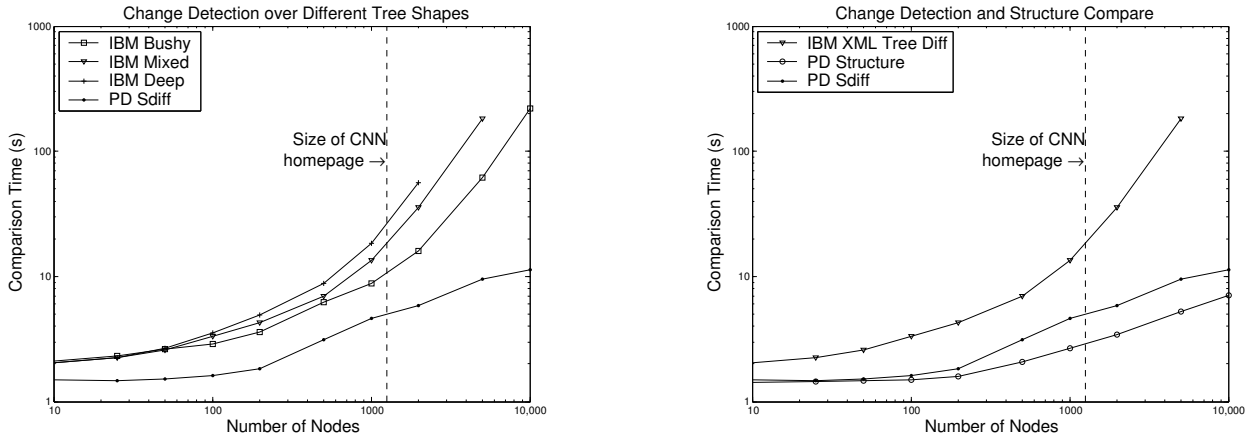
**Experiment 4: Page Digest Generation, Revert, and Load Time**

This experiment examines the time required to convert a Web document from its native format to the Page Digest for various document sizes. Figure 5(a) shows the performance of loading both parsed and pre-computed Page Digest encodings. For applications that can make use of cached data, such as change monitoring applications, caching Page Digest representations of pages is a clear improvement in storage space and load times that does not sacrifice any flexibility: the source HTML can always be reconstructed from the Page Digest if needed.

In addition, the graph shows the time needed to create a Page Digest contrasted with the time needed to construct an in-memory DOM tree of the same document. This test compares the JTidy [14] DOM parser for HTML DOM tree generation, the Xerces [20] SAX parser for HTML DOM, and the Page Digest generator which also uses the Xerces SAX parser. Although the SAX based HTML to Page Digest converter outperforms the error-correcting DOM parser, loading the native Page Digest stored on disk demonstrates the greatest performance gain since it requires minimal parsing to return an in-memory tree representation.

We note that the standard SAX to DOM converter performs slightly better that the Page Digest generator. This is due to the minimal overhead needed to organize the Page Digest after parsing, a cost which the DOM generator does not incur. However, this performance hit is mitigated by the substantial savings in traversal time afforded by the Page Digest, as shown in Experiment 1.

Figure 5(b) shows the performance of our implementation of Algorithm 2, which converts a document's Page Digest back to its native format. Note that unlike the discussion of Algorithm 2 presented in




(a) (b)

Figure 6: Page Digest Performance Comparison with IBM Corporation XML TreeDiff

Section 3.4, our implementation converts the original document exactly and preserves type, attribute, and content information. We confirm our algorithmic analysis and show that the time required to convert a Page Digest to its native form is linearly proportional to the size of the input document, and does not depend on the tree shape.

## 5.3 Sdiff Experiments

We tested Sdiff's performance relative to other document comparison approaches. Our goal with the performance tests was to emphasize the benefit of separating structure and content in Web documents, which is a core component in the Sdiff design.

**Experiment 5: Page Digest Change Detection**

Figure 6(a) compares Sdiff's performance with another tree difference algorithm, IBM Corporation's XML TreeDiff [12]. The trials for both tools cover our three test data sets: bushy, deep, and mixed. Note that the graphs are plotted in a log log scale. Also, although we ran all three tests for Sdiff and the Page Digest, we have only plotted the data for the mixed data set as the performance of the Page Digest trials were not significantly affected by changes in tree shape, with the result on this graph of all three lines being plotted in approximately the same place. For clarity, we omit the Page Digest deep and bushy lines.

Figure 6(b) emphasizes the Page Digest's ability to do meaningful document comparisons along different facets of the document. The IBM trial and the Page Digest Sdiff trial show the performance of the two tree

difference algorithm. Applications that only require structural comparison, however, can realize a significant performance increase by using the Page Digest, which the Page Digest Structure trial reveals.

**Experiment 6: Comparison with GNU Diff Algorithm**

Figure 7 shows a comparison between the performance of Sdiff and the standard GNU diff algorithm [10] in Java [9]. This experiment is designed to show Sdiff's performance compared with a traditional text based diff algorithm. We used the Java implementation to eliminate the confounding factor of different implementations from the experiment.

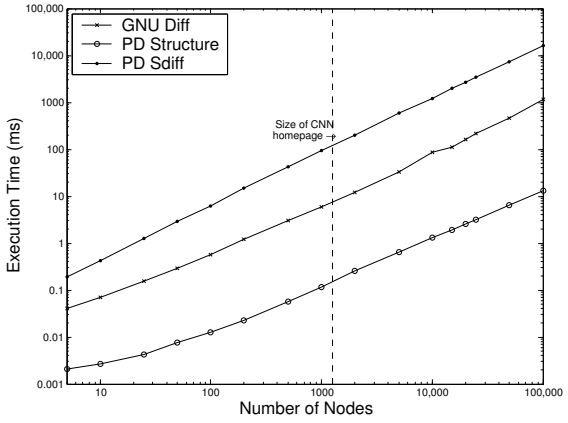



Figure 7: Page Digest Performance Comparison with GNU Diff

The results of this experiment show that GNU diff performs better than the full Sdiff algorithm but performs worse than the Sdiff structural change detection. We note that GNU diff is a text based algorithm that does not account for the tree structure or semantics of the documents it is comparing. Sdiff uses the structurally-aware Page Digest, so it can detect structural modifications very quickly. Sdiff also provides more control over the semantics of document comparison than is possible with GNU diff, making it more appropriate for Web services that operate on tree structured documents. Many Web services requires change detection that is specific to a single aspect of a document—such as tag names, attribute values, or text content. The GNU diff algorithm detects any changes whether they are interesting to the service or not.

# 6 Conclusion

We have introduced Page Digest, a mechanism for efficient storage and processing of Web documents, and shown how a clean separation of the structural elements of Web documents from their content promotes ef-

ficient operations over the document. Our experimental results show that Page Digest encoding can provide an order of magnitude speedup when traversing a Web document and up to a 50% reduction in the document size. We have seen the impact of Page Digest on the performance of the Sdiff information monitoring Web Service and shown how Page Digest's driving principle of separation of data and content provides the ability to perform very efficient operations for Web documents. Our analytical and experimental results indicate that the Page Digest encoding can play a foundational role in large scale Web services by enabling efficient processing and storage of Web data.

Our research on Page Digest continues along two dimensions. On the theoretical side, we are interested in studying the formal properties of Page Digest and its role in enhancing tree-based graph processing. On the practical side, we are interested in deploying the Page Digest encoding into a wider range of enabling technologies in the electronic commerce domain, including Web service discovery through similarity and Web page compression for efficient storage of Web data.

# References


[1] D. T. Barnard, G. Clarke, and N. Duncan. Tree-to-tree correction for document trees. Technical Report 95-372, Department of Computing and Information Science, Queen's University, Kingston, January 1995.

[2] A. Boldt. Cyclic redundancy check. http://www.wikipedia.org/CRC, 2002.

[3] A. Z. Broder. On the Resemblance and Containment of Documents. In *Proceedings of Compression and Complexity of SEQUENCES 1997*, 1997.

[4] D. Buttler, L. Liu, and C. Pu. A fully automated object extraction system for the world wide web. *Proceedings of IEEE International Conference on Distributed Computing Systems*, April 2001.

[5] Canoe, Inc. http://www.canoe.com, 2002.

[6] S. S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. pages 26–37, 1997.

[7] H. Dobbertin, A. Bosselaers, and B. Preneel. *RIPEMD-160: A Strengthened Version of RIPEMD*. 1996.

[8] FIPS 180-1. Secure Hash Standard. Technical report, U.S. Department of Commerce/N.I.S.T., National Technical Information Service, April 1995.

[9] S. D. Gathma. Java GNU Diff. http://www.bmsi.com/java/#diff, 2002.

[10] M. Haertel, D. Hayes, R. Tamllman, L. Tower, P. Eggert, and W. Davison. The GNU diff program. Texinfo system documentation, Available by anonymous ftp at `prep.ai.mit.edu`.

[11] R. Hamming. Error detecting and error correcting codes. *Bell System Tech. J.*, 29:147–160, April 1950.

[12] IBM Corporation. IBM XML TreeDiff. http://www.alphaworks.ibm.com/formula/xmltreediff, 2002.

[13] Y. R. Latifur Khan, Lei Wang. Change Detection of XML Documents Using Signatures. *WWW2002, Workshop on Real World RDF and Semantic Web Applications*, May 2002.

[14] A. Quick, S. Lempinen, A. Tripp, G. L. Peskin, and R. Gold. Jtidy. http://lempinen.net/sami/jtidy/, 2002.

[15] R. Rivest. The MD5 message digest algorithm. Technical report, Internet DRAFT, 1991.

[16] R. L. Rivest. The MD4 message digest algorithm. *Proc. CRYPTO 90*, pages 303–311, 1990.

[17] D. Rocco, D. Buttler, and L. Liu. Sdiff. Technical Report GIT-CC-02-59, Georgia Institute of Technology, 2002.

[18] D. Shasha and K. Zhang. Approximate tree pattern matching. In *Pattern Matching Algorithms*, pages 341–371. Oxford University Press, 1997.

[19] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, third edition, 1996.

[20] The Apache Foundation. http://xml.apache.org, 2002.

[21] World Wide Web Consortium. Well Formed XML Documents. http://www.w3.org/TR/REC-xml#sec-well-formed, 2000.