2012

An Implementation of a Dynamic Partitioning Scheme for Web Pages

Timothy Arndt
Cleveland State University, t.arndt@csuohio.edu

Ben Blake
Cleveland State University, benblake@csuohio.edu

Brian Krupp

Janche Sang
Cleveland State University, J.SANG@csuohio.edu

Publisher's Statement

Original Published Citation

Arndt, T., Blake, B., Krupp, B., Sang, J. (2012). An Implementation of a Dynamic Partitioning Scheme for Web Pages. International Journal of Computer Science Issues, 9(3), pp. 37-46.

An Implementation of a Dynamic Partitioning Scheme for Web Pages

Timothy Arndt, Ben Blake, Brian Krupp and Janche Sang

[1] Department of Computer and Information Science, Cleveland State University, Cleveland, Ohio 44118, USA

Abstract

In this paper, we introduce a method for the dynamic partitioning of web pages. The algorithm is first illustrated by manually partitioning a web page, then the implementation of the algorithm using PHP is described. The method results in a partitioned web page consisting of small pieces or fragments which can be retrieved concurrently using AJAX or similar technology. The goal of this research is to increase performance of web page delivery by decreasing the latency of web page retrieval.

Keywords: *Web Browser, Partitioning, Performance, PHP, Concurrency.*

1. Introduction

There has been much research done in the area of improving web performance by methods such as caching static content, pre-fetching web content and differencing and merging. However, with caching of static content the dynamic content of the page's performance doesn't improve. Also with pre-fetching, if the algorithm makes an incorrect decision on the future content to be requested, resources are wasted on requesting that content and processing that content.

Our approach to decreasing web retrieval latency will utilize existing standards and protocols to partition content within a page at the source and allow the partitions, or fragments, of the web page to be processed in parallel to improve web page delivery performance. This concurrent web page retrieval can be done using AJAX or some similar technology. The partitions or fragments in our implementation are created by looking for tags, though in general this could be done in any number of ways. Our general approach then is: web page fragmentation followed by concurrent retrieval of the fragments in order to minimize web page retrieval latency.

In the next section of this paper we will briefly review related work in the improvement of web page delivery performance. Section 3 will demonstrate out methodology for partitioning of a web page by the manual partition of an example page. This was the initial stage of our research and was done so that we could carry out performance testing on the fragmented web page to see if gains in performance were indeed possible. Having verified that this was in fact the case, section 4 describes the implementation of our partitioning method in a dynamic partitioning system using PHP. Conclusions are given in section 5.

2. Related Work

There has been a considerable amount of research in improving web page delivery performance. Some of the more recent and common research in this area has been in prefetching web content and caching of static content [3], [6], [7]. Caching, which has been implemented in web browsers for quite some time, has been coupled with proxies to allow caching to be done at an organizational level for better predictability.

One hybrid method that was proposed by Huang and Hsu [1] defined a method to mine popular sites using a prediction-based buffer manager that resides in front of a proxy to both cache and prefetch web pages. This method combines both caching and prefetching and removes the requirement for extra software to be installed on a user's machine.

A different approach proposed by Pons [5] used the Markov-Knapsack method to perform prefetching of web content by using the current web page and a Knapsack selector to determine the web objects to request. This model uses a server to keep track of prefetched pages, and pages that have been prefetched after.

An approach that focuses on improving crawling performance proposed by Peng, Zhang, and Zuo [4] looks at segmenting the web pages into relatively smaller units to expand the reach of crawling by navigating through irrelevant content to reach more important content. This approach takes one page that may be irrelevant as a whole and divides it up to find relevancy in a particular partition.

Finally, Jevremovic et al. [2] propose a Differencing and Merging System (DMS). DMS makes use of structural similarities which may exist between web pages and retrieves the difference between a previously fetched web page and the web page it now wants to retrieve. A model is developed in which the web server and browser maintain a history of web pages and differences and the web browser requests the minimum difference from the server in order to improve performance by sending the least amount of data over the network.

3. Manual Partitioning

To get an idea of the performance gains with dynamic partitioning and future design considerations, we created a sample page that contained several candidate partitions using the tag. We put a nested tag in as well as we expect we will come across nested partitions to see what would be the best approach of handling them. Now in the design of the framework, we are not restricted to tags, but will use them as an example as they are the predominant container tag in newer CSS design. After a page has been partitioned, we foresee the concurrent retrieval of those partitions using a technology like AJAX. That is reflected in the discussion in this section.

3.1 Approach

Looking at a sample of the code, we see some standalone tag as well as some nested tags where we outlined those areas:

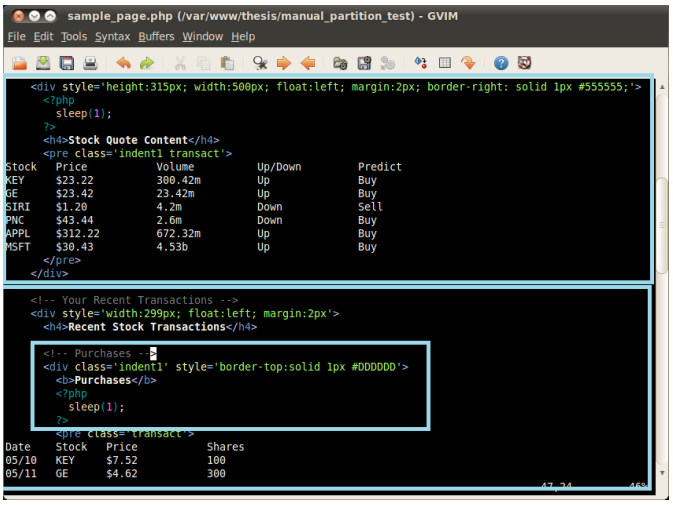

Fig. 1 Sample code.

Which, after rendering, produces the following site where we again outlined the different partitions:

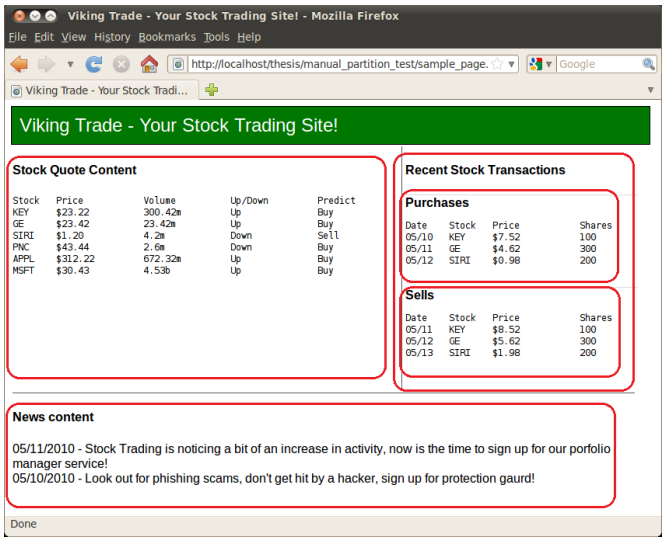

Fig. 2 Rendered site.

To do the manual partition, so that the partitioned content stands alone, there are two approaches we can use as shown in the next two subsections.

3.2 Separate File Approach

One approach is to separate the content of that partition, and store it in a separate file where the browser would make a request directly to that file. We would use the id attribute of the tag as part of the name of the separated content, if no ID existed, we would create one and store it in the tag.

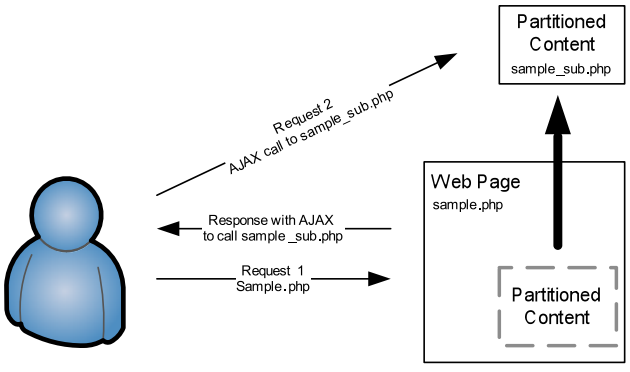

Fig. 3 Separate file approach.

From the above diagram, the framework would separate the content and store it in a separate file. The sample.php page would then include AJAX to call the partitioned content, so that the initial request to sample.php returns the

AJAX code to request the partitioned content, and the AJAX code would then place the response in the partitioned content area that it originated from.

3.3 Separate Method Approach

Another approach is to separate the content of that partition within the code from being executed by storing it in its own method. Then the browser as part of the AJAX code request for that method will execute in that particular page, and the results returned to the browser will be placed where the partitioned content was removed.

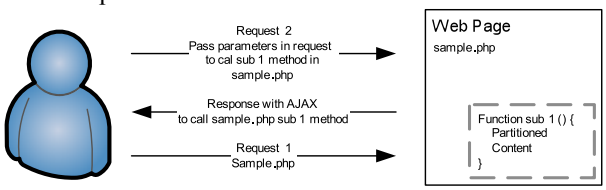

Fig. 4 Separate method approach.

Just like in the Separate File approach, we can use the ID of the tag that existed or the one we generated to name the function. Our research will focus on the separate file approach.

3.4 Parsing the Page

In either approach, when we parse the page, we need to keep track of the partition structure. To do this, we will create a basic tree, with a parent/child relationship to represent the nested tag structure. When parsing the page if we perform dynamic partitioning at the child and at the parent, we need to partition the child first, otherwise, when we take the partition of the parent out, it will include the child, and the code for the child will never be created.

Therefore as we walk our tree where each node represents a partition, we will need to check if there is a child, and if so go to the left-most child, and repeat. If there is no child, create the partition, move up to the parent, and delete the child where the partition was created. We will repeat this until there are no more elements in the tree except the root which would be the <html> tag.

An example of how this tree would look includes the following based on our example page is shown in figure 5.

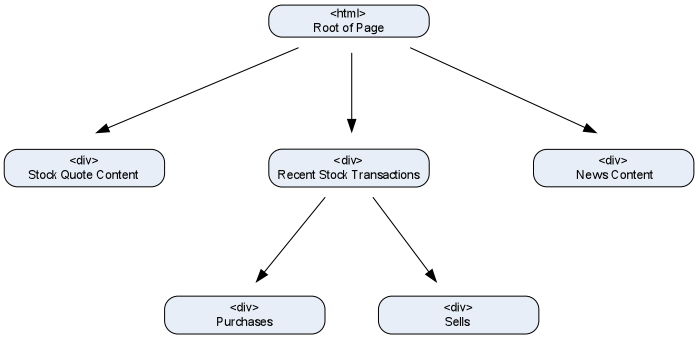

Fig. 5 Structure of example page.

So walking through this tree, we would start at the root, go to the Stock Quote Content, there are no children, so create the partition, and then remove that element from the tree, then go to the Recent Stock Transactions node, then Purchases, there are no children, so write out the partition, and remove the purchases node, at this state.

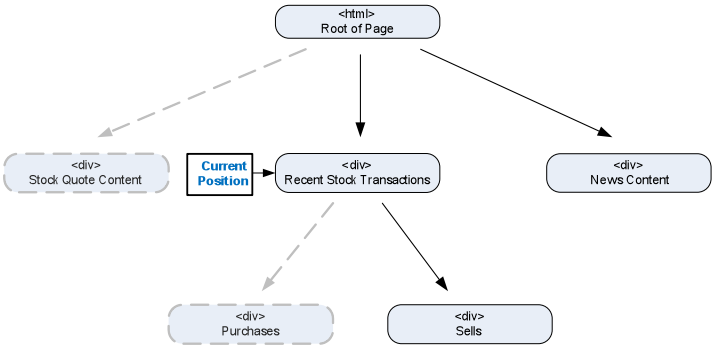

Fig. 6 Parsing the page.

Once we remove all nodes from the tree with exception to the root, we are done. In our example, when we assigned IDs to the tags, we had the mapping shown in table 1.

ID	Content
sub1	Stock Quote Content
sub2	Purchases
sub3	Sells
sub4	Recent Stock Transactions
sub5	News Content

Table 1 Mapping.

Performing the Separate File approach, we had the following files created: result_page.php, result_page_sub1.php, result_page_sub2.php and so on.

3.5 Performance Testing

We carried out an array of tests to verify whether our approach to increased performance was valid. We wanted to compare the retrieval time for the non-partitioned page (monolithic retrieval) versus concurrent retrieval of the partitioned page (fragmented retrieval). We set up software on the client side to generate the appropriate calls to the server. Our testing environment used a single server machine. With a single core machine, the performance gains were minimal. However, as would be expected with the concurrent approach we are aiming at, increasing the number of cores available on the server machine to two shows an appreciable performance gain, cutting the response time almost in half. This shows the validity of our approach.

4. Implementation of Dynamic Partitioning

In this section we discuss our implementation of the dynamic partitioning.

4.1 Designing the Parser

When looking at ways to do the dynamic partitioning, there were several approaches that we could take. One approach was to use a DOM parser that is available in PHP. We tested this approach first and found through our testing that the DOM parsers that are available are more suitable for traditional XML documents and not the kind of input that we would be working with where we will also have a mix of server side code and HTML.

Designing our own parser, we use regular expressions and build our own tree data structure to represent the nesting of elements and content. This allows us to easily walk the tree and extract elements for the dynamic partitioning.

Our parser will function as follows:

1. Create a ROOT element in the tree
2. Extract Content (optional), div tag then

   Remaining Content
3. Create Content as child of current element
4. If we hit end tag, go back to #2

If we were to parse the following HTML document:

```
<html>
  <body>
    Welcome
    <div id='msg'>
      Content before nested div
      <div id='nested'>
        Nested Content
      </div>
      Content after nested div
    </div>
    Goodbye
  </body>
</html>
```

We would get the following tree data structure:

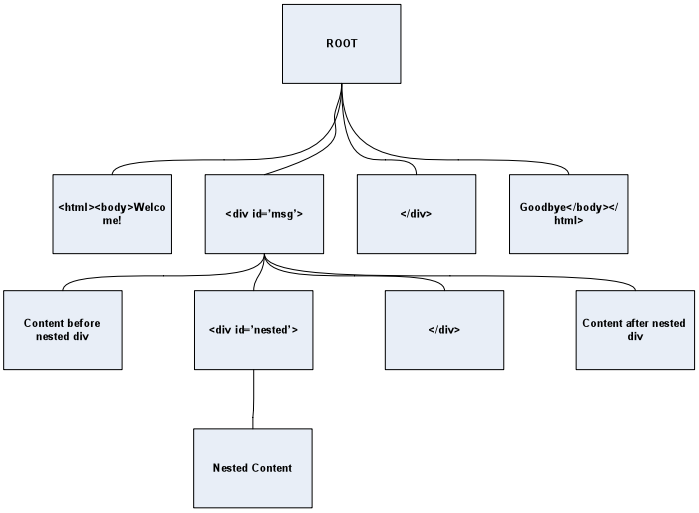

Fig. 7 Parsed tree data structure.

Once we have our tree data structure, we can then print out our HTML file by going to the left child that has not been accessed, printing its contents out, and repeating that process for each child that has not been accessed.

4.2 Implementation of Node Tree Structure in PHP

We built this implementation in PHP using an object oriented approach where we have a tree node object that can contain an array of children objects. These children objects would be other tree node objects. Other properties of this node contain an ID which would be used as the ID attribute in the div HTML tag, the tree node type which can be a nondiv, opendiv, and closediv, and the content of the node. Using the content of the node, if we walked the

tree from the root element to the left most element and repeat this for each untouched node, we would print out all the content in order.

The tree walk method that we designed allows us to pass a callback method that will be run on each node that the tree walk method reaches. This allows us to perform several operations on the tree with the same tree walk method.

4.3 ID Assignment

We need a unique ID for each partition. We designed the parser to use an existing ID if it exists, and if not, create a dynamic ID and increment it by one for each succeeding partition without an existing ID. This ID is then stored in the tree for quick retrieval as a property of the TreeNode class.

4.4 Separate File Approach

For this research, we implemented the separate file approach. To implement this approach, we had to come up with a way of storing the files effectively on the local filesystem. To do this, we create a directory where the parsed page is contained with a naming format of:

```
_<source_page>-dynpart
```

Within this directory, we store files based on the ID attribute of the Tree node. While we create these files however, we will more than likely have nested div tags:

```
<div id='1'>
  Content Before
  <div> Content Nested</div>
  Content After
</div>
```

In this scenario, we need two files for the content of the div tag with the ID of 1. One file will have "Content Before" as its content, the other will have "Content After". To work around this, we add a sub index to the file name. Following this approach, a div tag that has an existing ID would have the following file convention:

```
<id>_<sub_index>
```

And a dynamic generated ID would have the following file convention:

```
dynamic_partition_<dynamic
id>_<sub_index>
```

4.5 Pseudocode of Parser

The parser was created in PHP and used regular expressions within the code to grab tokens which were defined as content before tags, tags, content within tags, and content after tags and stored them in the tree such. The core pseudo code for the parser is as follows, note that comments start with the #.

```
# Create partition tree from
input file
Create root element for partition
tree and set as current node
While file has content
      If remaining content has a
      div tag, grab content up to
      div tag and div tag
            Add content before
            div to tree as child
            of current node
            If div tag is open
            div
                  Add tag as
            child of current node
                  Set current
            node to just created
            child
            If div tag is close
            div
                  Add as child
            node to parent of
            current node
                  Set current
            node equal to parent
            Set remaining content
            equal to content
            after div tag
      Else
            Add content of
            remaining file
            content as child to
            current node
Return tree to parser

# Walk tree and add unique
identifier for each div tag
Set current node equal to root
node
function walkTree
      If current node is an open
      div tag
            If current node
            doesn't have ID
            attribute
```

```
                  Assign dynamic
                  ID to node
      If current node has
      children
            Foreach child
                  walkTree of
      child

Prepare for dynamic partitioning
by creating filesystem for
separate file method using input
file name

# Dynamically partition the tree
function dynPartTree
      Foreach child of current
      node
            dynPartTree child
            If child type is
            within a div tag and
            is a nondiv type
                  Write child
                  content to
                  filesystem
                  using ID
                  if concurrent
                  AJAX library
                  has not been
                  included
                        Include
                        concurren
                        t AJAX
                        library
                        in child
                        content
                  Set content of
                  child =
                  concurrent AJAX
                  request for
                  child content
                  on filesystem

# Walk tree and print out
partitioned file to original file
Set current node equal to root
node
function walkTree
      Write to file node content
      If current node has
      children
            Foreach child
                  walkTree child
```

The actual code for this parser can be found in Appendix A.

4.6 Execution of Parser

The execution of the parser successfully performed dynamic partitioning of the page in a similar structure of the manual partitioned page, thus yielding the same performance results as the manual partition.

5. Conclusions and Future Research

In this paper we have described our approach to the web retrieval performance problem. First we partition a monolithic web page into fragments and then we retrieve those fragments concurrently. Our experiments show that are definite performance gains to be achieved using this approach, and we have shown that the web pages can be partitioned automatically, without manual intervention. This approach is especially appropriate where the web page contains dynamic content since in this case the caching techniques that others have developed are not relevant. In a future paper we will show how we can use AJAX to perform the concurrent retrieval and do performance testing on a prototype fragmentation/concurrent retrieval system.

Appendix A. – Dynamic Partition Parser PHP Code

```
#!/usr/bin/php -f
<?php
  /* First check if we want to get a
help for usage */
  if ($argc == 1 && $argv[1] == 'help')
{
    echo "\nUsage: dynPartPage.php
source_file\n\n";
    exit();
  }

  /* Then check for the arguments
  passed to the user, if the number of
  arguments equals the number of
  arguments equals the number of
  arguments we need, don't prompt the
  user, otherwise prompt the user for
  everything */
  if ($argc == 2) {
    // Get the input file
    $input_file = trim($argv[1]);
  } else {
    /* Prompt the user for a source
file */
    $input_file = getInput("Enter file
to convert");
  }
```

```
  $output_file = $input_file . "_new";

  // Perform input validation
  if (!file_exists($input_file))
die("Error: File ($input_file) does not
exist\n");

  // Grab the suffix of the file
  preg_match("/.*?\.(.*)/",
$input_file, $suffix);
  $suffix = $suffix[1];

  /* Create a tree from a source html
file */
  $root = createTree($input_file);

  /* Walk the tree, calling
addIdentifier callback */
  walkTree($root, 'addIdentifier');

  /* Prep dynamic partititon creates
the filesystem data structure needed */
  prepDynamicPartition($input_file);

  /* This does the magic and
dynamically partitions page */
  dynamicPartitionTree($root);

  /* Open the output file, and call
walkTree with callback of writeToFile
which will print the node content to
the file */
  $fh = fopen($output_file, "w");
  fwrite($fh, walkTree($root,
'writeToFile'));
  fclose($fh);

  /* Now that we made it this far,
rename the partitioned file and move
the newly created one on this one */
  $backup_file_name = $input_file .
".predynpart";
  $i=0;
  while
(file_exists($backup_file_name)) {
    $backup_file_name =
$backup_file_name . "_$i";
    $i++;
  }
  if (rename($input_file,
$backup_file_name)) {
    if (! rename($output_file,
$input_file)) {
      echo "Failed to move $output_file
to $input_file, exiting\n";
    }
  }
  else {
    echo "Failed to move $input_file to
$backup_file_name, exiting\n";
  }

  echo "Successfully created partition
page!\n\tStored pre-partition page at
$backup_file_name\n\tCreated dynamic
partition content in $dir_name\n\n";

  function getInput($prompt) {
    echo $prompt . " : ";
    return trim(fgets(STDIN));
  }

  function writeToFile($node) {
    global $fh;
    fwrite($fh, $node->content);
  }

  function printContentCallback($node)
{
    echo $node->content;
  }

  function
prepDynamicPartition($file_name) {
    global $dir_name;
    $dir_name = "_" . $file_name . "-
dynpart";
    if (is_dir($dir_name)) {
      $dh = opendir($dir_name);
      while (false != ($file =
readdir($dh))) {
        unlink($dir_name . "/" .
$file);
      }
      rmdir($dir_name);
    }
    mkdir($dir_name);
  }

  function dynamicPartitionTree($node)
{
    global $dir_name;
    global $first_pass;
    global $suffix;
    $children = $node->getChildren();
    foreach($children as $child) {
      dynamicPartitionTree($child);
      if ($child->isindiv && $child-
>type == "nondiv") {
      // Create our id and filename
```

```
      $id = $child->parent->id . "_" .
$child->parent->partition_count;
      $file_name = $dir_name . "/" .
$id . "." . $suffix;

/* Open file handler, and write the
content, and close the file handler */
      $fh = fopen($file_name, "w");
      fwrite($fh, $child->content);
      fclose($fh);

      /* Check if we made our first
pass, if we didn't, then add the script
content */
      if ($first_pass != "done") {
        $child->content = "<script
src='../common/js/concurrentAjax.js'
language='JavaScript'></script> " .
                        "<script> var
cAjaxRequestQueue = new Array();
</script>";
        $first_pass = "done";
       }
       else {
        $child->content = "";
       }

       $child->content .= "
        <span id='" . $id. "'></span>
        <script>

cAjaxRequestQueue[cAjaxRequestQueue.len
gth] = new cAjaxRequest('$file_name',
            function(response) {

       document.getElementById('$id').in
nerHTML += response;
            }
          );

cAjaxRequestQueue[cAjaxRequestQueue.len
gth - 1].doGet();
        </script>
       ";

       /* Increment the parent partition
count */
       $child->parent-
>partition_count++;
      }
    }
  }

  /* This will add a unique identifier
to each div tag */
  function addIdentifier($node) {
    /* Check to see if we have an open
div */
    if ($node->type == "opendiv") {
      /* If we do have an open div,
extract the ID attribute, and store it
in the object */
      $id_pattern =
"/.*?id\s*?=[\'\"](.*?)[\'\"].*?[\s\>]/
si";
      $nonid_pattern =
"/(<div)(.*)/si";
      if (preg_match($id_pattern,
$node->content, $matches))  $node->id =
$matches[1];
      // Else, add an ID
      else {
      preg_match($nonid_pattern, $node-
>content, $matches);
      $node->id = getUniqueId();
      $node->content = $matches[1] . "
id='" . $node->id . "' " . $matches[2];
      }
    }
  }

  /* This will create a unique ID and
return it */
  function getUniqueId() {
    global $id;
    if (! isset($id)) $id = 10000;
    else $id++;
    return "dynamic_partition_$id";
  }

  /* Function to walk tree in order the
way the elements were added, allows you
to pass the callback function */
  function walkTree($current_node,
$callback) {
    /* Call the callback on our current
node */
    $callback($current_node);

    /* Check if our current node has a
child, if so go through all of them */
    if ($current_node->getChildCount()
> 0) {
      /* Get list of children, and make
a recursive call to walkTree for each
child */
      $children = $current_node-
>getChildren();
      foreach($children as $child)
walkTree($child, $callback);
    }
  }
```

```
  /* Will need to have a separate node
called closediv, that will close a
previous tag */
  /* Tree node types nondiv, opendiv,
closediv  */
  class TreeNode {
    function TreeNode($type, $content,
$parent) {
      $this->type = $type;
      $this->content = $content;
      $this->parent = $parent;
      $this->children = array();
      $this->id = "";
      $this->partition_count = 0;

      if ($this->parent->type ==
"opendiv" || $this->parent->isindiv ==
true) $this->isindiv=true;
      else $this->indiv=false;
    }
    function addChild($child) {
      array_push($this->children,
$child);
    }
    function getChildCount() {
      return sizeof($this->children);
    }
    function getChildren() {
      return $this->children;
    }
  }

  /* This function returns a Tree
structure */
  function createTree($source_file) {
    /* Store the source file in a
single string */
    $source_file =
file_get_contents($source_file);

    /* Create root and store it in
current_node */
    $root = new TreeNode("root", "",
"0");
    $current_node = &$root;

    /* Keep going while the source file
contents are > 0 */
    while(strlen($source_file) > 0) {
      /* Check for any type of div tag,
have the s at the end of the reg ex to
span multiple lines */
      if
(preg_match("/(.*?)(<\/*?div.*?>)(.*)/s
i", $source_file, $matches)) {

      /* Add nondiv element which is
the content before the div */
      $current_node->addChild(new
TreeNode("nondiv", $matches[1],
$current_node));

      /* Check if we have a beginning
div, or an end div, first check for an
end div by checking for a / in the tag
       First check if we have an end by
checking if there is a / in the tag */
      if (preg_match("/.*?\/.*/si",
$matches[2])) {
        /* Add the close div to the
parent of this child */
        $current_node->parent-
>addChild(new TreeNode("closediv",
$matches[2], $current_node->parent));

        /* Point the current node to
the parent */
        $current_node = $current_node-
>parent;
      }
      /* Else we have an open tag, so
sent that to the current node, so we
can place the children underneath it */
      else {
        /* Create a temporary node, and
add it to the current node */
        $temp_node = new
TreeNode("opendiv", $matches[2],
$current_node);
        $current_node-
>addChild($temp_node);

      /* Store in current node the node
we just created since we will now be
adding whatever it contains to this */
        $current_node = $temp_node;
      }

      /* Store the remaining match into
the source file */
      $source_file = $matches[3];
      }
      /* Else, if we don't have any
divs left in the source, add to the
current node which should be the root
the left over content */
      else {
      $current_node->addChild(new
TreeNode("nondiv2", $source_file,
$current_node));
      $source_file = "";
      }
```

```
    } /* End of going through the
source file */

    /* Return the root node so we can
print out the tree */
    return $root;
  } // End of createTree function
?>
.
```

References

[1] Y.-F. Huang, and J.-M. Hsu. "Mining Web Logs to Improve Hit Ratios of Prefetching and Caching", Knowledge Based Systems, Vol. 21, 2008, pp. 62-69.

[2] A. Jevremovic, R. Popovic, D. Zivkovic, M. Veinnovic, and G. Shimic, “Improving Web Performance by a Differencing and Merging System”, International Journal of Computer Science Issues, Vol. 9, Issue 1, No. 1, January 2012, pp. 349-355.

[3] A. Kannanmmal, R. Padmanabhan, and R. Iyengar, “Web Cache Consistency Maintenance Through Agents”, in Proceedings of the Second International Conference on Communication Software and Networks, 2010, pp. 329-333.

[4] T. Peng, C. Zhang, and W. Zuo. "Tunneling Enhanced by Web Page Content Block Partition for Focused Crawling." Concurrency and Computation : Practice and Experience, 2007, pp. 61-74.

[5] A. P. Pons, "Improving the Performance of Client Web Object Retrieval." The Journal of Systems and Software, Vol. 74, 2005, pp. 303-311.

[6] L. Ramaswamy, L. Liu, and A. Iyengar, “Cache Clouds: Cooperative Caching of Dynamic Documents in Edge Networks”, in Proceedings of the 25th IEEE International Conference on Distributed Computing Systems, 2005, pp. 229-238.

[7] R. Sharman, S. S. Ramanna, R. Ramesh, and R. Gopal, “A Novel Hierarchical Cache Architecture for On-Demand Streaming on the Web”, ACM Transactions on the Web, Vol. 1, No. 3, 2007, pp. 23-49.

Timothy Arndt Received a Ph.D. in Computer Science from the University of Pittsburgh. He is currently an Associate Professor of Computer and Information Science at Cleveland State University. He is a Senior Member of the Association for Computing Machinery.

Ben Blake Received a Ph.D. in Computer Science from the Ohio State University He is currently an Associate Professor of Computer and Information Science at Cleveland State University.

Brian Krupp Received a Master’s Degree in Computer and Information Science from the Cleveland State University He is currently a Doctoral student in Software Engineering at Cleveland State University and Enterprise Security Architect at Key Bank.

Janche Sang Received a Ph.D. in Computer Science from Purdue University He is currently an Associate Professor of Computer and Information Science at Cleveland State University.