# A Shared Fragments Analysis System for Large Collections of Web Pages

Junchang Ma and Zhimin Gu

Department of Computer Science and Engineering,
Beijing Institute of Technology, Beijing 100081, China
swiftma@bit.edu.cn, zmgu@x263.net

**Abstract.** Dividing web pages into fragments has been shown to provide significant benefits for both content generation and caching. However, the lack of good methods to analyze interesting fragments in large collections of web pages is preventing existing large web sites from using fragment-based techniques. Fragments are considered to be interesting if they are completely or structurally shared among multiple web pages. This paper first gives a formal description of the problem, and then presents our system for shared fragments analysis. We propose a well-designed data structure for representing web pages, and develop an efficient algorithm by utilizing database techniques. Our system is unique in its shared fragments analysis for large collections of web pages. The system has been built and successfully applied to some sets of large web pages, which has shown its effectiveness and usefulness, and may serve as a core building block in many applications.

## 1 Introduction

The amount of information on the World Wide Web continues to grow at an astonishing speed, and the proportion of dynamic and personalized versus static documents is increasing day-by-day. To efficiently serve and deliver such dynamic and personalized content, several efforts have been made, among which fragment-based publishing and caching of web pages stands out. J. Challenger et al [1] presents a fragment-based publishing system for efficiently creating dynamic web content, which provides a method for web site designers to specify and modify inclusion relationships among web pages and fragments. Fragment-based caching features have already been offered by some products to optimize dynamic content processing on server side, e.g. BEA WebLogic [2], Oracle9iAS [3], Microsoft ASP.NET [4], and IBM WebSphere [5]. Many Java application servers allow programmers to mark a part of a page as cacheable using JSP tags. In ASP.NET such fragment can be explicitly put into a user control, which has its own cache parameters and can be included by pages or other user controls. [6] also uses tags to support fragment caching on a reverse proxy. Proxy+ [7] proposes an approach to enable ASP.NET fragment caching at enhanced web proxies. ESI [8] proposes to cache fragments at CDN stations to reduce network traffic and response time. ESI is a markup language that developers can

H. Bunke and A.L. Spitz (Eds.): DAS 2006, LNCS 3872, pp. 390–401, 2006.

use to identify content fragments for dynamic assembly at network edge servers. Therefore, only those non-cacheable or expired fragments need to be fetched from origin servers, thereby lowering the need to retrieve complete web pages and decreasing the workload of origin servers. These schemes have been shown to accelerate dynamic content generation and reduce network latency.

While performance benefits are important factors to consider, issues such as engineering complexity are of even higher priority. Fragment-based solutions typically rely on the web administrator or the web page designer to manually fragment the pages on the web site. Manual markup of fragments is both labor-intensive and error-prone, and may require considerable reengineering effort. Furthermore, the identification of fragments by hand becomes unrealistic and infeasible as the number of the existing web pages approaches tens of thousands. Thus there is a growing demand for techniques and systems that can automatically detect interesting fragments in web pages, and that are efficient and scalable enough for large collections of web pages.

In this paper, we consider fragments to be interesting if they are completely or structurally shared among multiple web pages, and we present a system that can automatically analyze shared fragments that are cost-effective for fragment-based techniques in large numbers of web pages.

The outline of the paper is as follows: in the next section we present a formal description of the problem. In section 3, we describe the fragments analysis system in detail. In section 4, we evaluate the system. In section 5, we discuss related work. In section 6, we conclude.

## 2 Problem Definition

Up to now, we have discussed the problem of shared fragments analysis loosely. In this section, we define it precisely. First, we introduce some definitions and notations.

*Definition 1. (Nodes Relations)* Two nodes in HTML DOM [9] tree are called *similar* iff all the following conditions are satisfied:

- They have the same type and name.
- They both or neither have attributes and children node.
- They have the same set of attribute names (if have).
- Their children nodes lists (if have) have the same length, and all the pairs of children node with the same index in the lists are *similar*.

They are called *equal* iff they satisfy all the following conditions:

- They are *similar*.
- They both or neither have values, if have, their values are same.
- They have the same attribute value for each of the attribute names (if have).
- All the pairs of children nodes (if have) with the same index in their parent's children nodes list are *equal*.

Both definitions are recursive with the base condition being that leaf nodes have no children node and can be compared directly.

*Definition 2. (Fragment)* For a web page $d$, let $T(d)$ denotes the HTML DOM tree of $d$, a *primitive fragment* of $d$ is defined as a node in $T(d)$ of type Element, and two or more adjacent primitive fragments with the same parent are called a *composite fragment*. Both *primitive fragment* and *composite fragment* are called *fragment*. Let $FN(d)$ be the number of fragments in $d$, associate each fragment in $d$ with a distinct number ranging from 1 to $FN(d)$, which is called its *fragment id*. Let $f(d, x)$ denotes the fragment of *fragment id* $x$ in $d$, and the set of all the fragments in $d$ is denoted by $F(d)$, $F(d) = \{f(d, x)|x \in [1, FN(d)]\}$.

*Definition 3. (Fragments Relations)* Two primitive fragments are called *similar/equal* iff their corresponding nodes are *similar/equal*. Two composite fragments are called *similar/equal* iff they have the same number of primitive fragments and all the pairs of them are *similar/equal*. Let $similar(f, f')/equal(f, f')$ denote two fragments $f$ and $f'$ are *similar/equal*. Fragment $p$ is called an *ancestor fragment* of fragment $f$, or equivalently, $f$ is called a *descendant fragment* of $p$, iff $p$ directly or transitively contains $f$. The set of all the ancestor fragments of $f$ is denoted by $PF(f)$, and the set of all the descendant fragments of $f$ is denoted by $DF(f)$.

The *similar* relation defined above captures the characteristics of *structurally* shared fragments, and the *equal* relation captures the characteristics of *completely* shared fragments. Having discussed the necessary definitions and notations, we now present the problem formally.

*Problem Statement.* Given a set of web pages $D$, $D_i \in D$, where $i \in [1, |D|]$, and $AF(D)$ denote the set of all the fragments in $D$, $AF(D) = \bigcup_{i=1}^{|D|} F(D_i)$. Two relations $R_{cs}$(*Complete Share Relation*) and $R_{ss}$(*Structural Share Relation*) are defined on $AF(D)$: $R_{cs} = \{(f, f') \in AF(D) \times AF(D)|equal(f, f')\}$,$R_{ss} = \{(f, f') \in AF(D) \times AF(D)|similar(f, f')\}$. It is easy to see that both of them are equivalence relations. Fix a parameter $M(M > 1)$ called *Minimum Shared Number*, a fragment $f$ in $AF(D)$ is called a *complete shared fragment* iff $|[f_{R_{cs}}]| \geq M$, where $f_{R_{cs}}$ is the equivalence class of $f$ on $R_{cs}$, and $f$ is called a *maximal complete shared fragment* iff it is a *complete shared fragment* and satisfies one of the following conditions:

- $\forall f' \in [f]_{R_{cs}}, \neg\exists p \in PF(f')$ and $p$ is a *complete shared fragment*.
- $\exists H \subseteq [f]_{R_{cs}}, f \in H, |H| \geq M, \forall f' \in H, \neg\exists p \in PF(f')$, and $p$ is a *maximal complete shared fragment*.

The definition is recursive with the first condition as the base condition. *structural shared fragment* and *maximal structural shared fragment* can be defined similarly. Let $CSF(D)$ denotes the set of all the *maximal complete shared fragments* in $AF(D)$, and $SSF(D)$ denotes the set of all the *maximal structural shared fragments* in $AF(D)$. Similarly to $R_{cs}$ and $R_{ss}$, two equivalence relations $R'_{cs}$ and $R'_{ss}$ are defined: $R'_{cs} = \{(f, f') \in CSF(D) \times CSF(D)|equal(f, f')\}$,$R'_{ss} = \{(f, f') \in SSF(D) \times SSF(D)|similar(f, f')\}$. The problem is to find the partition of $CSF(D)$ induced by $R'_{cs}$ and the partition of $SSF(D)$ induced by $R'_{ss}$.

## 3 Shared Fragments Analysis

In this section, we present the system for the problem defined in section 2. The working process of the system is divided into the following three steps:

1. Parse the given web pages one by one, and store their digest information needed for the analysis into database. We propose a data structure that is suitable for storing in database and efficient for fragment analysis, and provide the steps to transform web pages, which are detailed in section 3.1.
2. Find maximal shared fragments. This step is divided into two sub-steps. First, find the maximal shared *primitive* fragments. Second, merge the *primitive* fragments to find the maximal shared *composite* fragments. It's feasible because any set of maximal shared *composite* fragments can be viewed as multiple sets of maximal shared *primitive* fragments. We provide detailed explanation on the algorithms and implementations in section 3.2 and 3.3.
3. Collect share information. Statistics about fragments and web pages are collected in this step, e.g., size and popularity of fragments, number of shared fragments included by a web page, and the proportion of shared parts contained in a web page. We will not further discuss this step because it is application-specific and relatively easy in terms of implementation.

The system can, although can't simultaneously, analyze both *complete* and *structural* shared fragments. The processes are basically same, and the differences are explained at section 3.1.

### 3.1 Data Structure

A well-designed data structure for representing web pages is critical to efficient shared fragments analysis in large collections of web pages. The most popular document model is the Document Object Model (DOM) [9]. However, the memory-based DOM tree structure is infeasible for analysis of large collections of web pages, besides, the nodes of DOM tree don't contain sufficient information needed for efficient fragments analysis. These motivate us to store the fragments in DOM tree with augmented information into database, utilizing the mature database techniques in handling large relational data.

The DOM tree of a reasonably sized HTML page may have a few thousand elements. To limit the overhead of storing the elements and exclude very small segments of web pages from being detected as shared fragments, we introduce a parameter MIN_FRAG_SIZE (Minimum Fragment Size), which specifies the minimum size of the *primitive fragment* to be stored. *Composite fragments* are not stored because they can be composed of primitive fragments. Each fragment is stored in the database 'digest' table with *some* fields explained below. For explain convenience, *fragment* normally means the stored fragment in the following paragraphs.

Now we discuss *what* fields about a primitive fragment are stored in database.

First, some basic information about a fragment is stored. Concretely: (1) ID, the primary key of 'digest' table (2) DocID, the identifier of the web page this fragment belongs to (3) FragID, *fragment id* of this fragment. When a DOM tree is being traversed in pre-order, each fragment is numbered an increasing integer starting from 0 (4) StartLineNum, StartColumnNum, EndLineNum and EndColumnNum, the number of the start line, start column, end line and end column of the fragment in the web page respectively.

Second, the relations among fragments are stored. Concretely: (1) ParentFragID, the FragID of the fragment that directly contains this fragment (2) SiblingFragID, the FragID of the (supposed) next sibling fragment of this fragment (if it hasn't), which has an obvious but important property, namely, *a fragment is a descendant fragment of the current fragment iff they share the same DocID and its FragID is between the FragID, exclusive, and the SiblingFragID, exclusive, of the current fragment.*

Third, information for efficiently comparing fragments and detecting maximal shared fragments are stored. Concretely: (1) HashLow and HashHigh, the lower and higher 8 bytes of the MD5 hash value of the fragment. When finding *complete shared fragments (CSF)*, the hash method is DOMHASH [10], and fragments with the same hash value can be safely called *equal*. While when finding *structural shared fragments (SSF)*, the text data of Text node and attribute value of Attr node are not taken into consideration for hash, and fragments with the same hash value can be safely called *similar* (2) CompleteSize, the size of the data participating in the DOMHASH when finding *CSF*, which represents the actual size of the fragment (3) StructuralSize, the size of the data participating in DOMHASH in spite of *CSF* or *SSF* is currently being detected. *Similar fragments* may have different *CompleteSize* but are certainly have identical *StructuralSize*. *StructuralSize* has an important property, namely, *if a fragment f is of maximal StructuralSize and is a complete/structural shared fragment, then it is a maximal complete/structural shared fragment.* This property is based on the following observation. For each fragment $f'$ that is equal/similar to fragment $f$, $f'$ has identical StructuralSize with $f$, so $f'$ is also of maximal StructuralSize, if there exists a fragment that is an ancestor fragment of $f'$, its StructuralSize must be greater than the maximal StructuralSize, which is impossible.

Having discussed *what* information is stored, we now present the process to *get* the information and store it in database. Given a set of web pages, for each of them, the process is divided into four steps. First, associate each web page with a DocID (Document Identifier). Second, transform the web page to its DOM tree. The DOM parser used by us is based on CyberNeko HTML Parser [11], however, it has been revised so that it can provide the StartLineNum, StartColumnNum, EndLineNum and EndColumnNum fields about a fragment. Third, augment the DOM tree with MD5 hash, CompleteSize and StructuralSize. MD5 hash and StructuralSize are different based on *CSF* or *SSF* is being detected. Finally, assign FragID, ParentFragID and SiblingFragID to fragments whose StructuralSize not less than *MIN_FRAG_SIZE*, and insert them into database.

Since the web pages are converted one by one, the total time needed for this step increases linearly with the number and the total size of the given web pages. Moreover, the maximal memory consumption depends only on the largest web page. Thus, this step is very scalable and can effectively handle large collections of web pages.

### 3.2 Primitive Fragments Analysis

The process of primitive fragments analysis can be roughly divided into two steps. First, delete non-shared fragments. Second, find maximal shared primitive fragments by repeatedly using the property of StructuralSize.

A fragment is called a *non-shared fragment* iff the number of fragments having share relation with which is less than $M$ (*Minimum Shared Number*). Deleting these fragments can facilitate and speed up the shared fragments analysis. The deletion can be done by grouping all the fragments by hash value and deleting the fragments that are belong to the group whose member count less than $M$. The pseudo SQL statement of fragments selection is "select ID from digest group by HashLow, HashHigh having count (*)< $M$", and fragments deletion is "delete from digest where id in (list of IDs separated by comma)".

After the first step has been done, all the left fragments are shared fragments. According to the property of StructuralSize, the fragment of maximal StructuralSize is a maximal shared fragment, so are the fragments having the same hash as it, and they constitute an equivalence class of maximal shared fragments. To get all the fragments in the class, the hash value of the fragment of maximal StructuralSize is first obtained by the SQL statement "select HashLow,HashHigh from digest order by StructuralSize limit 1", and then all the fragments having the specified hash value are obtained by the pseudo SQL statement "select ID, DocID, FragID, ParentID, SiblingFragID, CompleteSize, StartLineNum, StartColumnNum, EndLineNum, EndColumnNum from digest where HashLow=specified_hash_low and HashHigh=specified_hash_high". To speed up the queries, indexes for StructuralSize and HashLow are created on 'digest'table.

Once obtaining an equivalence class, it is recorded into database. Two kinds of information are recorded, one is class-level information, and the other is the information of the fragments in the class. Concretely, class-level information is recorded into 'class' table with fields: (1) ClassID, the primary key of 'class' table (2) AvgSize, the average CompleteSize of the fragments in the class (3) ShareNum, the number of fragments in the class (4) LocHash, which will be introduced in the merging process. Information about each fragment in the class is recorded into 'msfrag' table with fields: (1) MSID, the primary key of 'msfrag' table. (2) ClassID, the identifier of the equivalence class this fragment belongs to (3) DocID, FragID, SiblingFragID, CompleteSize, StartLineNum, StartColumnNum, EndLineNum and EndColumnNum.

After recording the equivalence class, the fragments in the class need to be removed from further consideration. In addition, since all the descendants of them cannot be maximal shared fragments, they also need to be removed. All the descendants of a fragment $f$ can be easily found by the pseudo SQL state-

ment "select id from digest where DocID=f.DocID and FragID between f.FragID and f.SiblingFragID". To speed up this query, a multi-column index for DocID, FragID on 'digest' table is created. After removing, an originally shard fragment may not be any more because some of the fragments shared with it are removed for they have ancestors that are maximal shared fragments. So, in the later iteration, the fragments of maximal StructuralSize may not be maximal shared fragments, whereas, their descendant fragments may be. To avoid this complexity, an extra step to delete non-shard fragments can be added in this step, however, which is too expensive simply for this target. We choose to examine whether the detected fragments are real maximal shared fragments, and if not, do not record them, and delete them but remain their children.

To improve database update performance, all the above insertions and deletions are batched, and all the important queries are accelerated by indexes. So, by utilizing the database techniques, this step can handle large numbers of fragments.

### 3.3 Merging Primitive Fragments

The optimized algorithm to merge maximal shared *primitive* fragments into maximal shared *composite* fragments is still an open issue. However, we have the following observations:

Let $C_1, C_2, \ldots, C_M$ denote the ClassIDs of $M(M > 1)$ equivalence classes of primitive shared fragments, if each of the classes have $N(N > 1)$ fragments, and the list of all their fragments sorted by DocID and FragID is denoted by $LF(C_1, C_2, \ldots, C_M) = (f_{1,1}, \ldots, f_{M,1}, f_{1,j}, \ldots, f_{M,j}, f_{1,N}, \ldots, f_{M,N})$. If $\forall f_{i,j}, i \in [1, M], j \in [1, N], f_{i,j}.ClassID = C_i$ and $\forall f_{i,j}, f_{i+1,j}$, $i \in [1, M-1], j \in [1, N]$, $f_{i,j}.SiblingFragID = f_{i+1,j}.FragID$. Then $\forall j \in [1, N]$,$f_{1,j}$,$f_{2,j}$, $\ldots$,$f_{M,j}$ can be merged into one fragment. This is illustrated in Fig. 1.

Let $d(f_{i,j})$ denotes the DocID of fragment $f_{i,j}$, $p_{i,j}$ denotes the ParentFragID of $f_{i,j}$, and $DPL(C_i)$ denotes the sorted $d(f_{i,j})$,$p(f_{i,j})$ list of all the fragments belonging to class $C_i$, i.e. $DPL(C_i) = (d(f_{i,1}), p(f_{i,1}), \ldots, d(f_{i,N}), p(f_{i,N}))$. In the above conditions, all the adjacent fragments share the same DocID and

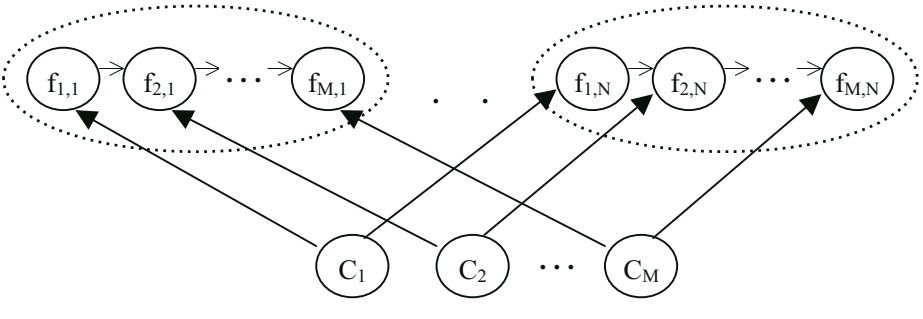



**Fig. 1.** $C_1, \ldots, C_M$ can be merged into one class, $f_{1,j}, \ldots, f_{M,j}$ can be merged into one fragment

ParentFragID, hence $DPL(C_1) = \ldots = DPL(C_M)$, which can be used as heuristic information when finding equivalence classes for merging. To ease the task of finding such classes, when each class is being recorded in primitive fragments analysis, the 32 bits Rabin [12] hash value of its $DPL$ is stored in the LocHash field. The reason to choose Rabin hash is that it has rare conflict and can be implemented efficiently [12].

The first step of merging primitive fragments is to get the set of LocHash for merging, which can be done by the SQL statement "select LocHash from class group by LocHash having count(*)> 1". Then, for each *LocHash* in the set, get the corresponding list of fragments for merging, this can be done by the SQL statement "select msf.* from class c, msfrag msf where c.ClassID=msf.ClassID and c.LocHash=*LocHash* order by DocID, FragID". To accelerate this query, two indexes are created, one for LocHash on 'class' table and the other for ClassID on 'msfrag' table. Afterwards, try to merge the selected fragments into one class. If succeed, delete old classes and fragments and insert merged class and fragments into database. Otherwise, try to merge part of the fragments. The later case is illustrated in Fig. 2. Due to space limitations, detailed implementations are skipped.

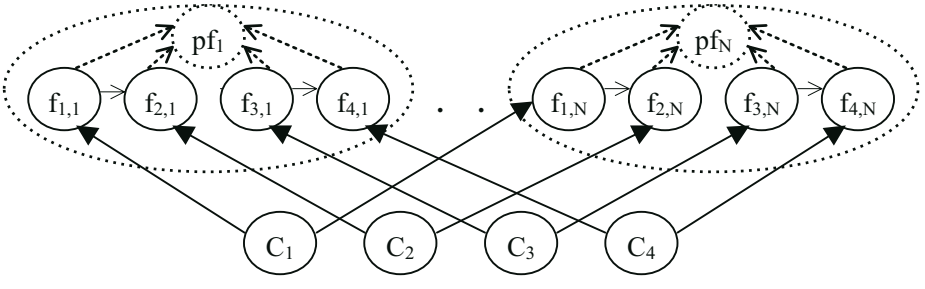



**Fig. 2.** Classes $C_1, C_2, C_3, C_4$ have the same LocHash and can't be merged into one, but $C_1$ can be merged with $C_2$, and $C_3$ can be merged with $C_4$

In this merging process, database queries are accelerated and updates are batched. In the worst case, this step takes much time, which may occur when one web page has a long list of adjacent maximal shared primitive fragments but none of them can be merged, however, it's very rare in practice. So, this step can also handle large numbers of fragments.

## 4 Evaluation

In this section, we first evaluate the performance of the system, and then present some fragment-level web content characteristics. The evaluations are performed on many large collections of web pages, which were downloaded from known web sites using GNU Wget (http://www.gnu.org/software/wget/wget.html). Due to space limitations, our discussions are restricted to the sina (www.sina.com.cn) and yahoo (www.yahoo. com) web sites. sina and yahoo are the most popular

web site in china and US respectively (reported in http://www.cwrank.com and http://www.comscore.com/metrix/). The sina data set contains 130,672 files and 1846M bytes, and the yahoo data set contains 7,379 files and 195M bytes. The parameter M (*Minimum Shared Number*) and *MIN_FRAG_SIZE* are set to 2 and 256 bytes respectively. For each data set, two experiments are performed, namely, *structural* shared fragments analysis (denoted by suffix _s) and *complete* shared fragments analysis (denoted by suffix _c). The four experiments are denoted by sina_c, sina_s, yahoo_c, and yahoo_s respectively.

### 4.1 Performance

Table 1 provides a synopsis of the time consumed of the four experiments. The total time needed for sina_c, sina_s, yahoo_c, yahoo_s is 366, 295, 42, 41 minutes respectively. Considering that the number and total size of the web pages are very large, and the task is inherently complex, the time consumed is very reasonable. Moreover, it can be seen that the total time needed approximately increases linearly with the number and total size of web pages, which is reasonable and acceptable. So, the system is expected to be able to effectively handle larger set of web pages.

**Table 1.** Breakdown of Analysis Time. Experiments Configuration: Intel Celeron CPU 2.4GHz, 512MB RAM, 40GB IDE DISK, Windows 2000 OS, MySQL 4.1.8 Database.

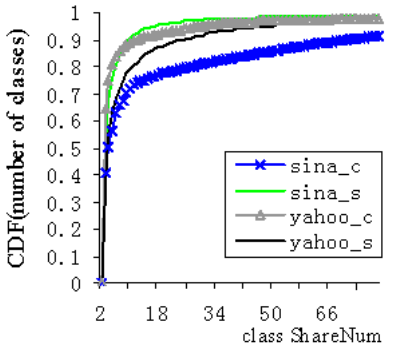
| Steps | sina_c | sina_s | yahoo_c | yahoo_s |
|---|---|---|---|---|
| Transforming Web Pages (s) | 13,254 | 12,490 | 2,248 | 2,198 |
| Primitive Fragments Analysis (s) | 8,703 | 5,189 | 263 | 284 |
| Merging Primitive Fragments (s) | 32 | 12 | 12 | 6 |
| *Total (s)* | 21,989 | 17,691 | 2,523 | 2,488 |

The total time is divided into three parts: Transforming Web Pages, Primitive Fragments Analysis, and Merging Primitive Fragments. The transforming step takes the most time among the three steps. Fortunately, which is very scalable, since it simply parses web pages one by one. Time for merging is insignificant, one reason may be that our merging algorithm isn't optimized and doesn't find all the worthy fragments. Time for primitive fragments analysis depends on the concrete characteristics of web pages, and the further analysis is skipped.

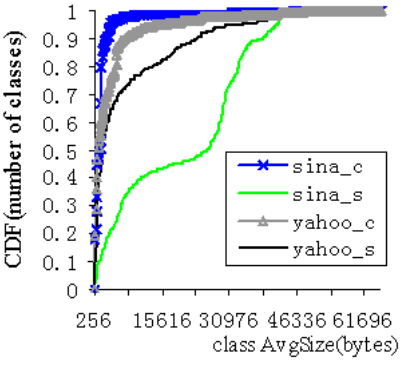
### 4.2 Fragment-Level Web Content Characteristics

Having discussed the performance, we now present some of the web content characteristics detected by our system, which are illustrated in Fig. 3. It can be observed that the structural shared fragments analysis has obvious different characteristics from the complete shared fragments analysis except in a). However, many common behaviors can also be observed.

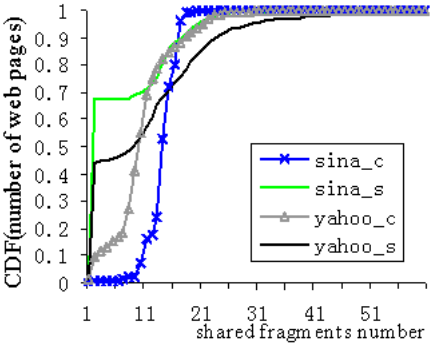
It can be seen from d) that a large percentage of web pages contain a great quantity of shared portions, and only a few web pages contain a small percentage



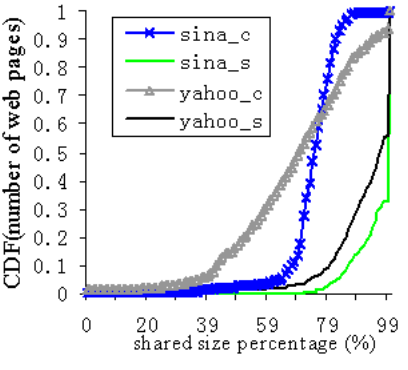

(a) class number distribution versus class ShareNum




(b) class number distribution versus class AvgSize




(c) web page number distribution versus shared fragments number




(d) web page number distribution versus shared size percentage

**Fig. 3.** Various CDF(Cumulative Distribution Function) for different experiments collected from the analysis result. a) CDF of the number of classes versus class ShareNum. b) CDF of the number of classes versus class AvgSize. c) CDF of the number of web pages versus the number of shared fragments contained in web pages. d) CDF of the number of web pages versus the size percentage of shared portions in web pages.

of shared portions. Besides, it can be observed from c) that the number of shared fragments found in a large proportion of web pages is small, and only a fraction of web pages contain large number of fragments. This indicates that our system is appropriate to be used as an automatic fragment tool for fragment-based techniques.

It is clear from a) and b) that a large percentage of classes have relatively small ShareNum and AvgSize, while a few classes have large ShareNum and AvgSize. It should be noted that, since web content characteristics are not the focus of this paper, our findings are very preliminary. However, further studies based on

the algorithm proposed in this paper are expected to find some underlying laws that can provide help to various researches.

## 5 Related Work

There has been significant work in identifying web objects that are identical [13], but they work at the granularity of entire pages. Various detection techniques for identical or similar code portions in source files have been proposed [15], which are related to our research on shared portions analysis, but the input to their algorithms are source code files which are not tree-like structures, and their line-based or function-based approaches are unfit for tree-like HTML pages. Numerous work on different aspects of analysis of web pages have been proposed, exemplified by discovering and extracting objects from web pages [14]. However, none of their work addresses the problem of shared fragments analysis.

The work by Lakshmish Ramaswamy et al [16] is the most related to our research. They also discuss the problem of shared fragments analysis in web pages for fragment-based caching. However, our work differs from theirs on three major aspects:

1. We give a formal description of the problem, and have considered the condition of composite fragments, while they don't.
2. We propose to store augmented DOM trees to database, and utilize database techniques to handle large numbers of web pages, while they develop a memory-based tree structure and algorithm to analyze shared fragments. Their solution may be more efficient than us when the number of web pages is small, but is infeasible for large collections of web pages.
3. They rely on the shingles fingerprinting [13] method to compare the relations between DOM nodes. Shingles is popular in estimating the resemblance of documents, however, which is based on random sampling techniques and improper for small texts that are popular in HTML nodes. While we rely on the semantic information of HTML structure and MD5 Hash to compare their relations, which is more proper.

## 6 Conclusions and Future Work

This paper makes several contributions:

- We give a formal description of the problem of shared fragments analysis. We have considered both *primitive* and *composite* fragments, and both *complete* and *structural* share relations.
- We propose a well-designed data structure for representing web pages, and provide the steps to transform the web pages. The structure is efficient for fragments analysis, and the transforming steps are scalable.
- We present an efficient algorithm for shared fragments analysis. We divide the algorithm into two steps, and describe the implementation steps and optimization strategies by utilizing database techniques.

– We evaluate the system through a series of experiments, showing its effectiveness in handling large set of web pages and usefulness in fragment-based techniques and studies of web content characteristics.

Our system is unique for its shared fragments analysis and ability to handle large numbers of web pages. However, the merging algorithm has not been optimized, and we will work on this issue in the near future. In addition, based on this system, we also plan to develop refactoring tools for assisting the adoption of fragment-based techniques, and study web characteristics at fragment granularity.

## References

1. J. Challenger, etc.: A Publishing System for Efficiently Creating Dynamic Web Content. Proceedings of INFOCOM'00, Mar.2000.
2. BEA WebLogic Server. http://www.bea.com/products/weblogic/server/.
3. Oracle9iAS. http://www.oracle.com/appserver/.
4. Microsoft. Caching Architecture Guide for .NET Framework Applications, 2003.
5. IBM WebSphere. http://www-3.ibm.com/software/webservers/appserv/.
6. Datta A, etc.: Proxy-Based Acceleration of Dynamically Generated Content on the World Wide Web: An approach and Implementation. Proceeding of ACM SIGMOD Intl. Conf. on Management of Data, Jun.2002, pp. 97-108.
7. Chun Yuan, Zhigang Hua, Zheng Zhang.: Proxy+: Simple Proxy Augmentation for Dynamic Content Processing, WCW'03.
8. ESI Consortium. Edge Side Includes. http://www.esi.org.
9. Document Object Model – W3C Recommendation. http://www.w3.org/DOM.
10. Network Working Group. Digest Values for DOM (DOMHASH). RFC2803, Apr.2000.
11. CyberNeko HTML Parser. http://people.apache.org/ andyc/neko/doc/index.html.
12. A.Z. Broder.: Some Applications of Rabin's Fingerprinting Method. In R. Capocelli, A. De Santis, and U. Vaccaro, editors, Sequences II: Methods in Communications, Security, and Computer Science, pages 143-152. Springer-Verlag, 1993.
13. A.Z. Broder. On the Resemblance and Containment of Documents. Proceedings of SEQUENCES-97, 1997.
14. D. Buttler and L. Liu. A Fully Automated Object Extraction System for the World Wide Web. In Proceedings of ICDCS'2001, 2001.
15. T. Kamiya, etc.: CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. IEEE Transactions on Software Engineering. Jul.2002.
16. Lakshmish Ramaswamy, Arun lyengar,Ling Liu, Fred Douglis.: Automatic Detection of Fragments in Dynamically Generated Web Pages. WWW2004, New York, May.2004.