Webstore: A Manager for Incremental Storage of Contents

Daniel Gomes
André L. Santos
Mário J. Silva

Webstore: A Manager for Incremental Storage of Contents

Daniel Gomes, André L. Santos, Mário J. Silva

Departamento de Informática

Faculdade de Ciências, Universidade de Lisboa

1749-016 Lisboa, Portugal

{dcg, als, mjs}@di.fc.ul.pt

November 2004

Abstract

This technical report details the design, implementation, and experimental results of Webstore, a manager for web data. Webstore addresses the requirements of warehousing applications that need to incrementally store and maintain contents gathered from the web. In web warehouses the existence of duplicated contents is prevalent. Webstore provides an efficient elimination of duplicates mechanism based on the analysis of the contents without requiring any additional meta-data. It provides unlimited growth of storage capacity, and distinct semantics of operation adaptable to various usage contexts. Our experiments showed that Webstore outperforms NFS by 68% in read operations and by 50% in write operations.

1 Motivation

The growing interest in web data mining raised the need for storage systems able to efficiently manage web data by addressing its specific characteristics [1, 2, 3, 4].

We consider web data as information gathered from the web. We subdivide it in two major classes: *contents* and *meta-data*. A content results from a download (e.g. an HTML file) and meta-data is information that describes the content (e.g. its size). This report presents Webstore, a system designed to support the incremental storage of contents. Webstore is part of a web data repository (XMLBase), which includes another component for managing the meta-data called Versus [5].

When two or more URLs refer to the same bytewise equal content, we say that they are duplicates. Several studies showed that duplication occurs frequently within a collection of web documents [6, 7, 8, 9, 10, 11]. There are many situations that lead to the existence of duplication on the web. For instance, publishers replicate documents or entire sites (mirrors) to ensure availability of the information. Many web designers build their sites by using other sites

Crawl ID	Finish Date	# urls (millions)	% dups. within the crawl	% dups. from the last crawl
1	15-07-2002	1.6	23	-
2	28-10-2002	1.2	21.4	7.4
3	21-03-2003	3.5	15.4	9.6
4	28-10-2003	3.3	10.9	18.5
5	16-06-2004	4.4	6.7	18.1

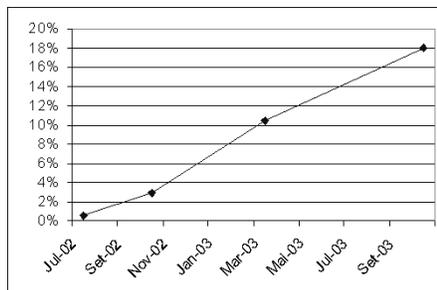Table 1: Duplication found in 5 crawls performed by the tumba! search engine.

Figure 1: Duplication found between crawl 5 and the previous crawls.

as templates. Commonly, web servers are configured to present default error pages when a given resource accessed through them, such as a database server is unavailable. These situations create duplicates of web pages that are very difficult to detect when one is gathering a large set of contents from the web. A collection of web contents built through successive crawls usually presents an additional number of duplicates, because a large number of contents do not change between the crawls. A proposed solution for this problem is to estimate the change frequency of pages based on historical data [12]. This may be used to reduce the probability of an incremental web crawler downloading a page that hasn't changed since the last crawl. Unfortunately, historical data for most web resources is not available. The problem of duplicates is also common in digital libraries caused by multiple deposits of a publication or by the existence of plagiarized contents [13, 14, 15, 16].

The tumba! search engine periodically crawls, indexes and archives textual contents from the Portuguese web, including sites from several Top Level Domains (TLD) [17, 18]. It was shown that the frequency of change is strongly related to the TLD of the host name [19, 20]. Therefore, we analyzed 5 crawls performed during the development of tumba!, in order to estimate the level of duplication within this collection of contents. We computed the MD5 digests [21] obtained for every content crawled, in order to detect duplicates.

In the presence of duplicated contents gathered from the web, it is not possible to automatically identify which URL is a replica and which is the original one. So, when several URLs point to the same content, we elected randomly one of the URLs as being referencing the original content and the remaining as duplicates. Table 1 summarizes the obtained results. For each crawl, we present

2

the date in which the crawl was finished, the total number of URLs crawled and the percentage of URLs that were duplicates (level of duplication). In the right-most column we present the percentage of URLs that referenced a content that already existed in the previous crawl. The first version of the crawler generated *crawl 1*, where *23%* of the contents were duplicates. We found that the number of duplicates within successive crawls has been decreasing due to consecutive optimizations of the crawler, such as the identification of duplicated hosts through the analysis of historical data We observed that, as the crawler performed more exhaustive crawls of the Portuguese web, the number of contents that already existed in the last crawl increased. These results suggest that the duplication within a collection of web documents is strongly related with the tools used to gather the contents. However, finding or developing a suitable tool to gather specific contents from the web, is usually a difficult task.

Figure 1 shows the percentage of URLs from the most recent crawl (*5*), which are duplicates from each one of the previous crawls. We can see that the occurrence of duplicates between crawls decreases with time. However, duplication between crawls distant in time is considerable, crawl 4 and 5 were performed with 8 months of interval, but there were still 18% duplicated contents. Our analysis strongly suggests that duplication would become a problem as the crawls become more frequent, causing a big waste of storage capacity.

The tumba! system was initially using the Network File System (NFS) [22] to access of the storage the contents crawled from the web. This was the cause of several problems. NFS imposes the configuration of operating system parameters of its clients in order to ensure its correct functioning. For instance, system clocks must be synchronized and operating system users must maintain the same identifier in all the machines hosting NFS clients. This became a big problem when we had to run crawlers on borrowed machines that were not under our administration. Besides, the crawlers had been unexpectedly loosing contents and suffering from crashes of the Java Virtual Machine, while they were storing contents on NFS volumes. We suspect that these errors were caused by failures of the NFS server when it is overloaded with requests from the crawlers.

So, there was a need for an easily manageable system, with extensible storage capacity and simultaneously addressing the problem of duplication. Webstore is composed by a set of autonomous storage nodes, with no central point of coordination. Its storage capacity can be extended by adding new nodes, without imposing major changes in the system. In addition to these features, common in distributed storage systems, Webstore provides a mechanism for fast detection and elimination of duplicates. Besides saving disk space, we show that this mechanism may also increase the storage throughput of contents.

This paper is organized as follows: the next section presents the requirements and the assumptions made in the design of Webstore. In Section 3 we introduce the data model of the system. Section 4 presents the operations provided for data management and the underlying algorithms. Section 5 describes the architecture and implementation of the prototype. In Section 6, we expose, interpret and compare experimental results. Finally, in Section 7, we present related work and in Section 8 we draw our conclusions and propose directions for future work.
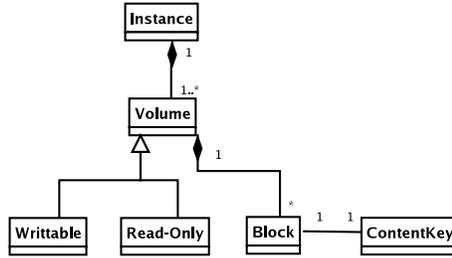
Figure 2: Webstore class diagram.

2   Requirements and Assumptions

Webstore must be expansible and maintain all the stored information available online. It assumes an underlying computer cluster, composed by several independent and possibly heterogeneous nodes that provide storage space. The storage nodes do not communicate among them. All the nodes are located on the same local area network and have similar distance costs to the clients. It is assumed that there are no malicious clients. These assumptions relieve the system from security and routing issues, that must be addressed by storage systems distributed over wide-area networks.

The addition of new storage nodes and the migration of contents from exhausted nodes to new ones with bigger storage capacity must be supported. We assume that these operations are sporadic and that new nodes added to the system bring an increased storage capacity.

The system tolerates faults of storage nodes, but it does not provide automatic recovery and full availability of the stored contents at the faulty nodes. A stronger tolerance model can be achieved at disk level, for instance, using random arrays of inexpensive disks (RAID), or, through a content replication schema defined by the clients. The application programming interface provides methods that enable applications to implement their own storage policies, in order to fulfill their requirements.

Webstore can detect and eliminate exact duplicates. The detection of contents partially duplicated is not addressed. When a node exhausts its storage capacity we admit that duplication of contents between nodes may occur.

Contents can be stored in compressed formats. Several types of compression must coexist within a node, so that contents with different formats can be compressed with a suitable algorithm. We assumed that the collections managed by Webstore contain millions of contents with similar probability of being read; therefore, caching mechanisms are not required.

The system software is platform independent and can run at application level without imposing changes in the configuration of the underlying operating systems.

3   Data Model

The data model relies on 3 main classes: *instance*, *volume* and *block* (Figure 2). The class names were inspired in concepts commonly used in database and file
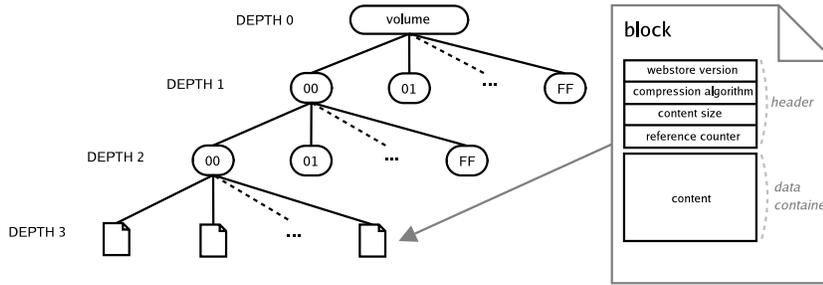
4

Figure 3: Storage structure of a volume: a tree holding blocks on the leafs

systems. The instance class provides a centralized view of Webstore to the clients. Each instance is composed by a set of volumes. A volume has an internal tree structure where it keeps the blocks. Each block keeps a content and related meta-data. The clients identify each content through a *contentkey*.

3.1 Contentkey

A *contentkey* is composed by two fields. The first identifies the volume where the content is stored. The second holds the signature of the content. The signature is the number obtained from applying a fingerprinting algorithm to the content.

When a content is stored, Webstore generates a contentkey and returns it to the client. In order to retrieve or delete a content from Webstore, the client must supply the corresponding contentkey. A contentkey can be converted to a string format, so that it can be easily handled by clients.

3.2 Block

The contents are stored on the leafs of the volume's storage tree in *blocks*. A block holds an unique content within the volume and has an associated reference counter that keeps track of how many times the content was stored. Therefore, several independent applications can share the same instance without interfering with each others processing of the volume's data.

A block (see Figure 3) is composed by a *header* and a *data container*. The header keeps meta-data about the instance and the held content. Each header of a block has the following fields:

- *webstore version* - a 32 bit integer that identifies the Webstore software version;

- *compression algorithm* - a string that specifies the algorithm used to compress the content;

- *content size* - an integer specifying the original content size, in bytes;

- *reference counter* - a 32 bit integer that keeps track of the difference between the storage and delete requests performed on the content.

5

The data container keeps the content. Webstore enables the usage of several compression algorithms, which can be chosen by the applications according to the characteristics of the content. For instance, a text compression algorithm is suitable for compressing a HTML document, but not for an image. As the compression algorithm is defined for each content when it is stored, the same instance may keep contents compressed in several formats.

3.3 Instance and volume

Each instance provides an independently managed storage area. An instance is composed by a set of volumes, each hosted in an independent storage node that provides disk space. Each volume is identified by a number, from 0 to the maximum number of volumes minus one that compose the instance.

Each storage node contains a volume where the contents are kept. A volume can be in one of two states: *read-only* or *writable*. A volume is *writable* if it accepts new contents and must be set to *read-only* when its storage capacity is exhausted.

The storage structure of a volume is a tree of containing *blocks* on its leafs. Figure 3 illustrates a storage structure with depth 3. The nodes within each level of depth are identified by numbers represented in hexadecimal format from 0 to FF. The tree depth can change within the volumes that compose an instance, according to the storage capacity of the node.

3.4 Block look-up

The location of a block within the volume tree is obtained by applying a function called *sig2location* to the content's signature. Assuming that the signature of a content is unique, two contents have the same location within a volume if they are duplicates. However, it is possible that duplicate contents exist in different volumes. Sig2location is defined as follows:

- Consider a volume tree with depth n and a signature with m bytes of length;

- The *(n - 1)* most significant bytes in the signature identify the path to follow in the volume tree. The ith byte of the signature identifies the tree node with depth i. The tree has a degree of 256 so that when looking up for a block, the search space is reduced by the maximum factor, as each byte of the contentkey is processed;

- The remaining bytes of the signature $(m\text{-}n\text{-}1)$ identify the block name on the leaf of the tree.

For example, considering a volume tree with depth *3* (see Figure 4), the block holding a content with signature *ADEE2232AF3A4355* would be found in the tree by following the nodes AD, EE and leaf 2232AF3A4355.

The contentkey that references an overflow includes the suffix in the signature.
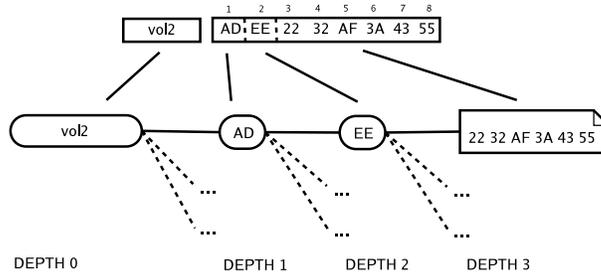
Figure 4: Decomposing a contentkey: the first field identifies the volume, the second identifies the nodes in the tree and the leaf block

4   Management

Webstore provides three basic operations to manage the contents: *store*, *retrieve* and *delete*. The elimination of duplicates is performed during the storage operation, so this is the most complex one. Next, we will detail the mechanisms used and the semantics for each operation.

4.1   Store

The detection of duplicates is performed during the store operation, ensuring that each distinct content is stored in a single block within an instance. When a client requests the storage of a content, Webstore performs sequentially a set of tasks:

1. Generate a signature s for the content to be stored;

2. Apply sig2location to the signature, to obtain the location l of the corresponding block;

3. Search for a block in location l, within the n writable volumes that compose the instance. This search begins by the volume identified with the number resulting from $s \ mod \ n$;

4. If a block is found, the content is considered to be a duplicate and it is not stored again. The reference counter is incremented and a contentkey identifying the existent block is returned to the client;

5. If a block is not found, the content is stored in a new block with location l in volume $s \ mod \ n$, and a contentkey referencing this block is returned to the client.

This way, while the configuration of volumes that compose the instance is not changed, if one tries to store the content again, Webstore detects a duplication in the first volume it searches. The algorithm proposed to eliminate duplicates between volumes, has costs proportional to the number of writable volumes that compose the instance. We assumed that a new node is added to the system when an existing one exhausts its storage capacity and it is set to read-only. This way,

the number of writable volumes remains constant, as well as the inherent cost of eliminating duplicates between them.

The mod-based policy used to determine the volume where to store a content divides the load equally among the volumes. Although, if the storage nodes are very heterogeneous, this policy may not be adequate. Webstore allows the clients to define the volume where to store each content. This way, other load-balancing policies can be defined at the application level by defining the volume where to store the content. This could be useful, for instance, to impose a higher workload on volumes with better throughput capacity.

4.1.1 Methods for detecting duplicates

Theoretically, if two contents have the same signature they are duplicates. However, fingerprinting algorithms present a small probability of collision. A collision occurs when the algorithm generates the same signature for two different strings [23]. Therefore, if two contents have the same signature we say that they are *potential duplicates*. Relying exclusively on the comparison of signatures to detect duplicated contents would cause some contents to be wrongly identified as duplicates and not stored. We call this situation a *fake duplication*.

The occurrence of fake duplicates can be reduced by comparing the sizes of the potential duplicates. Assuming that the probability of two contents having the same size is independent from the probability of fingerprint collision between them, if two contents have the same signature but different sizes, they must be fake duplicates. However, the success of this heuristic is highly dependable on the distribution of the content sizes within the data sets. So, the most reliable way to eliminate fake duplicates is through a bytewise comparison of the potential duplicates, in exchange for some performance degradation.

4.1.2 Store modes

When Webstore detects a fake duplicate, it creates an overflow block to keep the content and the client receives a contentkey to this block. In order to enable the usage of Webstore in various contexts, it supports three different modes for the store operation, according to the policy followed to detect fake duplicates:

- *regular* - Relies on the comparison of the sizes of the contents to detect fake duplicates. If two potential duplicates have different sizes, they are considered fake duplicates. Otherwise they are considered duplicates. Notice that if two potential duplicates have the same size, a fake duplication may occur. The regular mode is suitable when the occurrence of fake duplicates is tolerated.

- *compare* - Relies on size and bytewise comparison of the contents to detect fake duplicates. If two potential duplicates have different sizes or have the same size but are not byte equal, they are considered fake duplicates. A client should use the compare mode when the possibility of the occurrence of a fake duplicate is not admissible.

- *force-new* - There is not a distinction between potential and fake duplicates. An overflow is created to keep every potential duplicate. So, there isn't elimination of duplicates, since every content is stored in a new block.
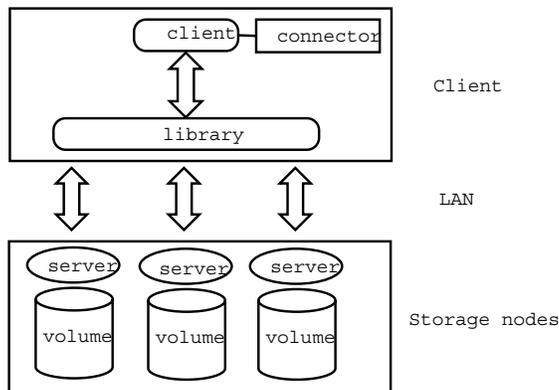
Figure 5: Architecture of the system.

The semantics of the store operation is defined for each invocation, so that the clients can determine different semantics according to the contents being stored. For example, when building a historical archive of a set of documents of great historical importance, one could use the *force-new* mode to store these contents and the *regular* mode for the rest.

4.2 Retrieve and Delete

A client retrieves a content by supplying the correspondent contentkey. Webstore decomposes the contentkey, identifies the signature of the content and the volume that hosts the correspondent block. The location of the block in the volume is obtained by applying sig2location to the signature. Finally, the content stored in the block is decompressed according to the algorithm specified in the block's header, and the content is returned to the client.

The delete operation is also invoked with a contentkey as argument. The location of the block is executed following the same process as for the retrieve operation. If the reference counter contained in the header of the block has value 1, the block is deleted. Otherwise, the reference counter is decremented.

Since the location of the content is determined by the contentkey, the volume where the content is stored is directly accessed, both for the retrieve and delete operations. Therefore, the performance of these operations is independent from the number of volumes that compose an instance.

5 Prototype

Webstore has a three-tier architecture (Figure 5). An instance is composed by a thin middleware library, the connector object and the volume servers that manage the volumes.

Clients access an instance through a connector object. The connector describes the volumes that compose the instance. A change in the composition of the instance, such as the addition of a new volume, implies an update of the connector. The clients execute operations through the invocation of meth-

9

```
<?xml version='1.0' encoding='UTF-8'?> <instance name="myWebstore"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="webstore-config.xsd">
  <volume name="vol1" state="writable" tree-depth="1"
    host="ren.fc.ul.pt" port="9999"/>
  <volume name="vol2" state="read-only" tree-depth="2"
    host="stimpy.fc.ul.pt" port="10000"/>
</instance>
```

Figure 6: The connector object implemented in XML format.

ods provided by the API library. The contents are transmitted between the library and the servers in a compressed format chosen for content storage in Webstore, to reduce network traffic and data processing on the server. Each volume server manages the requests and executes low-level operations on the volume it manages.

5.1   Implementation

Webstore can be easily installed and it is platform-independent. It was successfully tested on RedHat Linux 9.0 (kernel 2.4.20-8) and Windows 2000.

In the Linux implementation, the storage structure of a volume was implemented as a directory tree over the ext3 filesystem [24]. Each node of the tree is represented by a directory. The blocks are files residing at the leaf directories. These files are divided in two parts: a fixed length header and the data container. The header is written in ASCII format so that it can be human readable. We used a 64-bit implementation of the Rabin's fingerprinting algorithm [23] to generate content signatures.The only compression method supported in the current prototype is Zlib. A client can also store a content in its original format without compressing it.

The connector object was implemented as an XML file. An example is presented in Figure 6. The instance represented is composed of 2 volumes with different depths and only volume 1 is available for write. The server that manages volume 1 is listening on port 9999 of host *ren.fc.ul.pt*.

The library and the volume servers were written in Java using JDK 1.4.2. The communication between them is done through Berkeley sockets. A volume server is multi-threaded, launching a thread to handle each request. The server guarantees that each block is accessed in exclusive mode through internal block access lists.

6   Results

In this section, we present a set of 6 experiments ran on Webstore and compare its performance against NFS on Linux (nfs-utils-1.0.1-2.9). These experiments reproduce the typical usage of our web data storage system, hence we can evaluate Webstore as a replacement of NFS in the tumba! system. Besides, NFS is widely known and easily accessible, enabling reproducibility of our experiments. We used three machine configurations to host the clients and the NFS and Webstore servers:
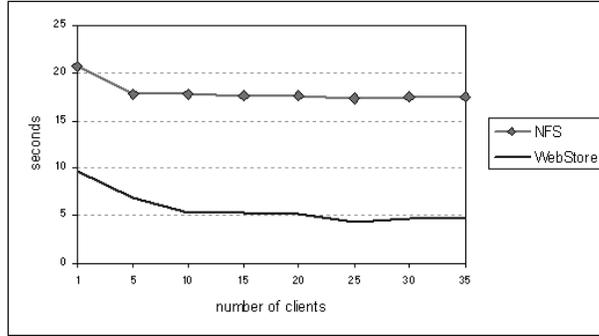
Figure 7: NFS read vs. Webstore retrieve.

- *Configuration 1*: Fujitsu-Siemens Primergy P250, with two 2.4 GHz Pentium IV Xeon CPUs, 4 GB PC133 SDRAM and five 73 GB Ultra160 SCSI 10000 rpm hard drives. The disks run in hardware RAID 1 (mirroring) using an Adaptec DAC960 controller. The operating system used is RedHat 7.3, kernel 2.4.20-20.7smp.

- *Configuration 2*: ASUS AP140R, with one 2.4 GHz Pentium IV CPU, 1.5 GB PC133 SDRAM and two 180GB IBM Deskstar IC35L180AVV207-1, IDE ATA 100, 7200 rpm hard drives. The disks are run in software RAID 5 (disk striping with parity) mode, using md driver 0.90. We used RedHat 9.0, kernel 2.4.20.8 on this configuration.

- *Configuration 3*: ASUS AP1720I5, with two 2.4 GHz Pentium IV Xeon CPUs, 2 GB DDR SDRAM and five 250 GB Western Digital WD2500JB-00FUA0, IDE ATA 100, 7200rpm hard drives. The five disks (4+1 spare) are run in software RAID 5 (disk striping with parity) mode, using md driver 0.9. We used RedHat 9.0, kernel 2.4.20.8smp for these machines.

We gathered a data set composed by 1000 distinct HTML pages with a total size of 19.2 MB from a crawl performed by tumba!.

In the first three experiments, we compared the performance of the three basic operations of Webstore (store, retrieve and delete), against NFS using a single volume. In experiment 4, we compared the performances of the store operation alternatives. Finally, we analyzed the scalability of the detection of duplicates between volumes (experiment 5) and its impact on the load-balancing between volumes (experiment 6).

6.1 Experiment 1: Retrieving

The experimental setup used in the following 3 experiments consisted in:

- five Configuration 1 machines, one Configuration 3 machine and one Configuration 2 machine, each one hosting five clients;

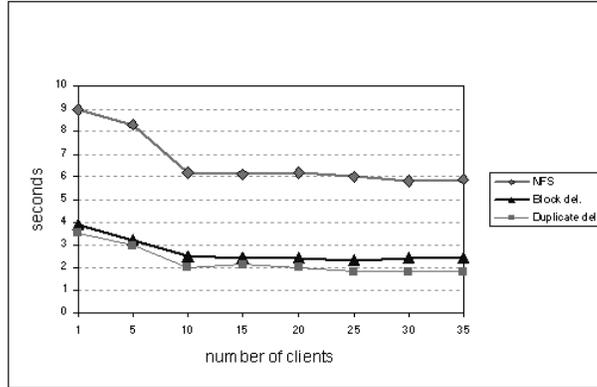- one Configuration 3 machine that hosted the NFS and Webstore volumes.

Figure 8: NFS remove vs. Webstore delete.

First, we loaded the volume server with the contents from the data set, using the compression option, and kept all the generated contentkeys. Then, we split the contentkeys among the clients and ordered them to retrieve all the correspondent contents. In each measurement, we added a new machine hosting 5 clients in order to stress the servers. The fastest machines were the last ones to be added to the cluster of clients.

For the NFS test, we copied the data set to a directory on the server machine and exported it to every client. Then we split the file paths among the clients and ordered them to read all the contents from the data set in parallel. Before each measurement, we re-mounted the NFS volume on every client machine to ensure that there was not caching of the files.

Figure 7 presents the total time that the clients took to read all the contents from Webstore and NFS. We found that Webstore is on average 68% faster than NFS for read operations. The total time for executing the task in Webstore remained constant for more than 10 parallel clients with Webstore and more than 5 clients with NFS. We believe that Webstore is faster than NFS because it transfers the whole content from the server at once, while NFS transfers data in 32 KB chunks. The tested NFS implementation relies on caching mechanisms to achieve performance for the read operation. However, we intentionally did not use this feature because we assumed that all the contents within a collection of documents are accessed with the same probability.

6.2   Experiment 2: Deleting

We loaded the data set into the Webstore and NFS volumes and split the references to the contents among the clients, as described in the previous experiment. We launched the clients that deleted all the contents from the data set. Then we loaded the data set into Webstore twice, creating a situation where all the contents were duplicates and relaunched the clients.

Figure 8 compares the total time that the clients took to delete all the contents from NFS and Webstore. Our system is on average 67% faster than NFS when deleting a duplicate, (only the reference counter is decremented in this case), and 60% faster than NFS when deleting the block. We believe that
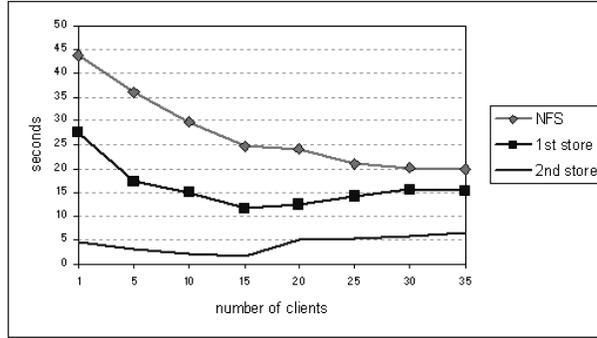
Figure 9: NFS save vs. Webstore regular store.

Webstore outperforms NFS because it uses a lighter protocol of communication and does not have the overhead of maintaining caches of files and associated meta-data such as permissions. Both the NFS and Webstore servers reached their maximum throughput with 10 clients.

6.3   Experiment 3: Storing

In this experiment we split the data set among the clients and launched them in parallel, with the mission of storing the data set in Webstore, and then in NFS. For each set of clients, we repeated the task twice in Webstore. The objective was to compare the times spent to create blocks to store new contents and to add references in the presence of duplicates. We used the regular mode for the store operation in Webstore. The NFS volume was exported with the *sync* option, which does not allow the server to reply to requests, before the changes made by the request are written to the disk (same behaviour as Webstore). The experiments on NFS had to be restarted several times due to JVM (Java Virtual Machine) crashes of the clients.

Figure 9 presents the total times taken to store the contents in Webstore and NFS. As expected, the store operation is faster for storing duplicates than for storing new contents, because the contents are not transferred from the clients to the volume servers and the creation of a new block is not required. Webstore seems to reach its saturation point with 15 clients, while NFS achieves it only with 25 clients. However, our system outperformed NFS on average by 50%, from 5 to 20 clients, and by 26% for more clients. For duplicated contents Webstore outperforms NFS by 82%.

6.4   Experiment 4: Semantics of the store operation

The setup for this experiment consisted in:

- one Configuration 3 machine hosting 10 clients;

- one Configuration 1 machine hosting the NFS and Webstore volumes.

The objective of this experiment was to measure the performance of the 3 different modes available for the store operation: regular, compare and force-new. We gradually increased by 20% the level of duplication within the data set
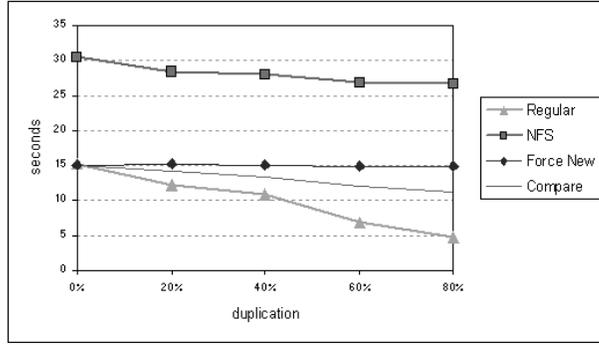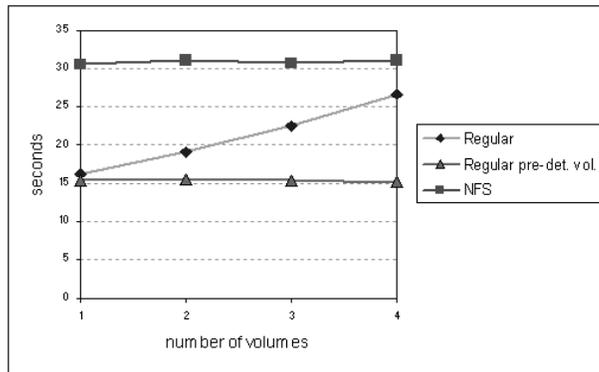
Figure 10: Semantics of the store operation.

Figure 11: Duplicates elimination scalability.

and launched 10 clients that stored the data set using each one of the modes. Figure 10 presents the results obtained. The semantic of the force-new mode is similar to the NFS write operation, given that it does not put any effort in eliminating duplicates. However, it took almost half of the time to finish the task.

As the level of duplication increased, the compare and regular modes presented the best results. The compare mode is slower than the regular, because it detects duplicates by comparing the contents byte-per-byte, while the regular mode compares only the sizes of the contents.

When there wasn't replication within the data set, all the three modes performed the same way. Therefore, we conclude that the overhead of detecting duplicates within a volume is insignificant. Besides saving disk space, we showed that the proposed mechanism for the elimination of duplicates increases the storage throughput of the system when it manages collections containing replicated contents.
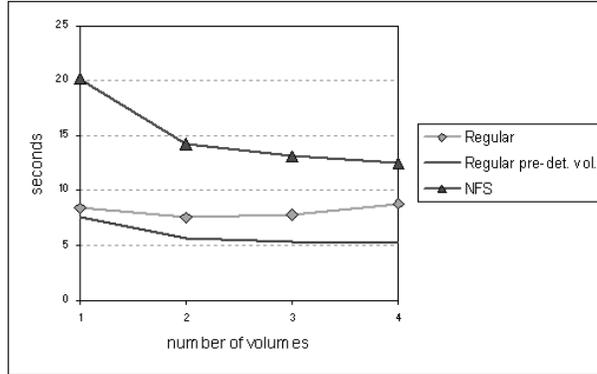
14

Figure 12: Load balancing between volumes.

6.5   Experiment 5: Duplicates detection between volumes

This experiment measured the impact of the elimination of duplicates between volumes in the performance of Webstore. We used the following setup:

- one Configuration 1 machine hosting one client;

- four Configuration 2 machines, each one hosting: an NFS and a Webstore volume.

In each run we added a new volume and ordered the client to store the data set. All the contents stored in the volumes were removed before each run. Firstly, we measured the total time spent to execute the task without eliminating duplicates between volumes. For this purpose, the Webstore client used the regular store option but chose the volumes where the contents were going to be stored. Then, we measured the total time spent by the Webstore client to execute the task without choosing the destination volumes, hence eliminating duplicates between volumes.

Figure 11 presents the obtained results. We observed that, without eliminating duplicates across volumes, Webstore outperforms NFS by 50%. As expected, when Webstore eliminates duplicates across volumes, the time spent to execute the task increases (by 15%) for each node that is added to the instance. Figure 11 suggests that with more than 5 volumes NFS would take less time than Webstore to execute the task.

We concluded that the elimination of duplicates across volumes should be used when the instance is composed by a small number of writable volumes. However, if the collection of contents kept in Webstore has a high level of duplication, the cost of eliminating duplicates could be compensated by the faster storage of duplicates, as it was shown in experiments 3 and 4.

6.6   Experiment 6: Load Balancing

This experiment was designed to evaluate the influence of load balancing among the volumes. We used the following setup:

15

- one Configuration 1 machine and one Configuration 3 machine, hosting two clients each;

- four Configuration 2 machines, hosting a NFS and Webstore volume each.

As in the previous experiment, the clients stored the data set in the volumes and we gradually incremented the number of volumes that composed the instance. Figure 12 presents the results obtained. We observed that NFS considerably reduced the total time of the task everytime we added a new volume. In the case of Webstore the time started increasing for more than 2 volumes, when we used the regular store. The reason for this was that the overhead of eliminating duplicates between volumes overwhelms the gainings from load balancing. When we pre-determined the destiny volumes, there was not elimination of duplicates between volumes, but the time remained almost constant from more than 2 volumes. We believe that most of the time was spent in network traffic. As the network time remains constant independently from the number of volumes that compose the instance, the effect of load balancing amongst the volumes becomes very subtle. Unfortunately, when we were compiling the results for this report, we could not have exclusive access to the network, so that we could obtain clearly defined experimental conditions for the observed times spent in network traffic.

7   Related Work

Modern databases can manage contents but their design is centralized and an increase of performance usually implies the migration of the system to more powerful hardware [25].

Versioning systems such as CVS [26] or Aegis [27] save disk space by using delta-storage. They assume that objects change in time maintaining a descendence tree under an unique identifier (file name). This assumption is not applicable to web contents because duplication occurs between contents with distinct identifiers (URLs).

Peer-to-peer file systems are designed to manage a large and highly variable set of nodes with small storage capacity, distributed over wide-area networks (typically the Internet). This raises specific problems and imposes complex intra-node communication protocols that guarantee properties such as security, anonimicity or fault tolerance [28, 29, 30, 31]. Webstore has a different scope because it assumes nodes with big storage capacity located within the same local-area network and rare changes on the set of storage nodes.

Lustre is an open source project that presents a storage and file system architecture suitable for very large clusters of Linux servers [32]. It presents impressive scalability and redundancy. However, it is platform dependent and requires patches on the operating system kernel to be installed. Webstore shares architectural aspects with network file systems [33, 34]. However, most of them are executed at the operating system kernel level and features are limited to a file system interface.

Regarding web research, in the Webase project the authors studied how to construct and maintain fresh a large shared repository of web pages [4]. The powerDB project developed a document search and indexing engine, based on a PC cluster, each of them running an off-the-shelf DBMS [35]. The research paper

of the popular search engine Google described a repository where web pages are stored sequentially in compressed packets [36]. Recently, Google presented its own file system [37]. Herodotus is an web archival system based on a Peer-to-Peer architecture [38]. The Internet Archive stores the data collected in large aggregate files [39].

Webstore differs from all the presented systems due to its innovative capability of detecting and eliminating duplicates at storage level.

8 Conclusions and Future Work

This report presented the design, implementation and experimental results of Webstore, a system specially designed for the management of document collections containing duplicates. Webstore is platform-independent and runs at the application level. It presents an innovative distributed mechanism to eliminate duplicates, while it also allows the explicit creation of replicas, enabling clients to define their own storage policies. The experimental results showed that our system outperforms significantly NFS in read, write and delete operations. The results also showed that the mechanism proposed for the elimination of duplicates may increase storage throughput. The source code of Webstore is available for download at `http://xldb.fc.ul.pt/webstore/`.

Webstore was included as a component of the tumba! search engine and it is currently being loaded with its previous crawls. In 2005, we intend to release to the public the first archive of the Portuguese web, managed by Webstore.

We will enhance Webstore with further compression algorithms. The inclusion of a compression algorithm that supports data mining over compressed text is currently in development [40]. This will enable searching of text snippets without uncompressing the whole content.

The system must be tested in further usage contexts, managing different types of data. Deeper research is needed to improve the detection of duplicates between volumes. The usage of distributed hash structures and the introduction of communication between the volume servers are also options under consideration.

9 Acknowledgements

References

[1] Catriel Beeri, Gershon Elber, Tova Milo, Yehoshua Sagiv, Oded Shmueli, Naftali Tishby, Yakov A. Kogan, David Konopnicki, Pini Mogilevski, and Noam Slonim. Websuite: A tool suite for harnessing web data. In *International Workshop on the Web and Databases*, pages 152–171, 1998.

[2] Cheng Kai, Yahiko Kambayashi, Seok Tae Lee, and Mukesh K. Mohania. Functions of a web warehouse. In *Kyoto International Conference on Digital Libraries*, pages 372–379, 2000.

[3] Hans-Jorg Schek, Klemens Bohm, Torsten Grabs, Uwe Rohm, Heiko Schuldt, and Roger Weber. Hyperdatabases. In *Web Information Systems Engineering*, pages 14–25, 2000.

[4] Junghoo Cho, Hector Garcia-Molina, Taher Haveliwala, Wang Lam, Andreas Paepcke, Sriram Raghavan, and Gary Wesley. Stanford webbase components and applications. Technical report, Stanford Database Group, July 2004.

[5] Daniel Gomes, João P. Campos, and Mário J. Silva. Versus: a web repository. In *WDAS - Workshop on Distributed Data and Structures 2002*, Paris, France, March 2002.

[6] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the web. In *Proceedings of the Sixth International conference on World Wide Web*, pages 1157–1166. Elsevier Science Publishers Ltd., 1997.

[7] Narayanan Shivakumar and Hector Garcia-Molina. Finding near-replicas of documents and servers on the web. In *Selected papers from the International Workshop on The World Wide Web and Databases*, pages 204–212. Springer-Verlag, 1999.

[8] Krishna Bharat and Andrei Broder. Mirror, mirror on the web: a study of host pairs with replicated content. In *Proceedings of the Eighth International Conference on World Wide Web*, pages 1579–1590. Elsevier North-Holland, Inc., 1999.

[9] Fred Douglis, Anja Feldmann, Balachander Krishnamurthy, and Jeffrey C. Mogul. Rate of change and other metrics: a live study of the world wide web. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[10] J. Mogul. A trace-based analysis of duplicate suppression in HTTP. Technical Report 99/2, Compaq Computer Corporation Western Research Laboratory, November 1999.

[11] Terence Kelly and Jeffrey Mogul. Aliasing on the world wide web: Prevalence and performance implications. In *Proceedings of the 11th International World Wide Web Conference*, Honolulu, Hawaii, May 2002.

[12] Junghoo Cho, Hector García-Molina, and Lawrence Page. Efficient crawling through URL ordering. *Computer Networks and ISDN Systems*, 30(1–7):161–172, 1998.

[13] Raphael A. Finkel, Arkady Zaslavsky, Krisztian Monostori, and Heinz Schmidt. Signature extraction for overlap detection in documents. In Michael J. Oudshoorn, editor, *Twenty-Fifth Australasian Computer Science Conference (ACSC2002)*, Melbourne, Australia, 2002. ACS.

[14] Narayanan Shivakumar and Héctor García-Molina. SCAM: A copy detection mechanism for digital documents. In *Proceedings of the Second Annual Conference on the Theory and Practice of Digital Libraries*, 1995.

[15] Sergey Brin, James Davis, and Héctor García-Molina. Copy detection mechanisms for digital documents. pages 398–409, 1995.

[16] Antonio Si, Hong Va Leong, and Rynson W. H. Lau. Check: a document plagiarism detection system. In *Proceedings of the 1997 ACM symposium on Applied computing*, pages 70–77. ACM Press, 1997.

[17] Mário J. Silva. Searching and archiving the web with tumba! In *CAPSI 2003 - 4a. Conferência da Associação Portuguesa de Sistemas de Informação*, Porto, Portugal, November 2003.

[18] Daniel Gomes and Mário J. Silva. Characterizing a national community web (accepted for publication). *ACM Transactions on Internet Technology*, 5(2), May 2005.

[19] Dennis Fetterly, Mark Manasse, Mark Najork, and Janet L. Wiener. A large-scale study of the evolution of web pages. In *Proceedings of the 12th International World Wide Web Conference*, Budapest, May 2003.

[20] Junghoo Cho and Hector Garcia-Molina. The evolution of the web and implications for an incremental crawler. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14*, pages 200–209, September 2000.

[21] R. Rivest. *RFC 1321 - The MD5 Message-Digest Algorithm*, 1992.

[22] B. Callaghan, B. Pawlowski, and P. Staubach. *RFC 1813: NFS Version 3 Protocol Specification*. Sun Microsystems, Inc., June 1995.

[23] M. O. Rabin. Probabilistic algorithm in finite fields. Technical Report MIT/LCS/TR-213, 1979.

[24] RedHat Inc. *Red Hat Linux 7.3 The Official Red Hat Linux Reference Guide*, chapter 5. RedHat Inc., 2002.

[25] Oracle9i. `http://www.oracle.com/ip/index.html?content.html`.

[26] B. Berliner. CVS II: Parallelizing software development. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 341–352, Berkeley, CA, 1990. USENIX Association.

[27] Peter Miller. Aegis is only for software, isn't it? http://aegis.sourceforge.net/auug96.pdf, 1996.

[28] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: the Oceanstore Prototype. In *Proceedings of the 2nd Usenix Conference on File and Storage Technologies (FAST'03)*, 2003.

[29] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66, 2000.

[30] Frank Dabek, Emma Brunskill, M. Frans Kaashoek, David Karger, Robert Morris, Ion Stoica, and Hari Balakrishnan. Building peer-to-peer systems with Chord, a distributed lookup service. pages 81–86, 2001.

[31] Karl Aberer, Philippe Cudré-Mauroux, Anwitaman Datta, Zoran Despotovic, Manfred Hauswirth, Magdalena Punceva, and Roman Schmidt. P-grid: a self-organizing structured p2p system. *SIGMOD Rec.*, 32(3):29–33, 2003.

[32] Peter J. Braam. *The Lustre Storage Architecture*. Cluster File Systems, Inc., April 2004.

[33] Ohad Rodeh and Avi Teperman. zfs: A scalable distributed file system using object disks. In *Proceedings of the 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03)*, page 207. IEEE Computer Society, 2003. zFS extends the research done in the DSF project.

[34] Randolph Y. Wang and Thomas E. Anderson. xfs: A wide area mass storage file system. Technical report, 1993.

[35] Torsten Grabs, Klemens Böhm, and Hans-Jörg Schek. A parallel document engine built on top of a cluster of databases - design, implementation, and experiences. Technical Report 340, Department of Computer Science, ETH Zurich, April 2000.

[36] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[37] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles, October 19-22, 2003, Bolton Landing, NY USA*. ACM, 2003.

[38] Timo Burkard. Herodotus: A peer-to-peer web archival system, 2002.

[39] Mike Burner and Brewster Kahle. WWW Archive File Format Specification, September 1996.

[40] N. Brisaboa, E. Iglesias, G. Navarro, and J. Paramá. An efficient compression code for text databases. In *Proceedings of the 25th European Conference on Information Retrieval Research (ECIR'03)*, LNCS 2633, pages 468–481, 2003.