Online Solutions for Scalable File Server Systems

Savio S.H. Tse
Computer Engineering Department
Bilkent University
Ankara, Turkey
sshtse@cs.bilkent.edu.tr

Abstract

We propose three online algorithms for scalable file server systems. A scalable file server is expected to provide rather stable services while the numbers of users, tasks, and data volumes keep increasing. One of the purposes of parallel and distributed approaches is to achieve scalability. Sufficient hardware resources are essential for good services; however, a good coordination of them is also indispensable, as parallel and distributed resources need to complement the shortages of each other, and it falls on the shoulders of the algorithmic and architectural designs. In this paper, we address the load balancing problem in scalable file servers.

The three online approximate algorithms proposed is for placing and deleting documents in a system of M distributed file servers located in a cluster in order to balance the loads and required storage spaces among all servers. In [7], we have proposed some algorithms without allowing re-allocation. In this paper, by paying the re-allocation cost, we have several improvements on some existing results.

1 Introduction

We tackle the problem of scalability in distributed file server systems in algorithmic ways. A scalable file server is expected to provide rather stable services while the number of users keeps increasing. One of the purposes of parallel and distributed approach is to achieve scalability. Sufficient hardware resources are essential for good services. However, its disadvantages are difficulties in scheduling and balancing the parameters of each distributed entity. Therefore, a good coordination of them is also indispensable, as parallel and distributed resources need to complement the shortages of each other. The qualities and quantities of resources are also expected to be scaled up gradually. In order not to sacrifice the availability for scalability, we design online efficient algorithms to coordinate all resources. (Offline solutions will sacrifice the availability of systems.) In this paper, we address the online problem of balancing load and storage space with the allowance of re-allocation. By online, we mean the time cost for each operation is reasonably bounded.

Our first result is an algorithm for placing documents into the heterogeneous server system. Heterogeneity is obviously an advantage for scalability, because there are less limitations on servers. We do not assume replication in this case. Suppose the heterogeneity among servers is reflected by the individual load and storage space bounds. Formally, the jth server will have bounds $p_l^j L$ and $p_s^j S$ for load and storage space, respectively, where L and S are the optimal bounds for load and storage space of each server, respectively, and $p_l^j, p_s^j > 2$, $p_l^j + p_s^j \geq 6$, and $j \in [1, M]$. The time for inserting one document is $O(S + \log M)$.

From another point of view, load balancing can easily be done by full and excessive replication. However, because of the high time-cost for carrying out replication and updating replicas, as well as the high storage-cost, we do not apply this technique in this paper. As discussed in [7], a high storage-cost will be translated to a time cost eventually. In this paper, the storage space is also a parameter needed to be balanced among the servers. Our second task is to bound the loads and storage spaces by $(k_l - \frac{1}{2})L$ and $k_s S$, respectively, in $O(\log M)$-time for each document insertion into a homogeneous server system, where $k_l, k_s > 2$, and $\frac{1}{k_l - 1} + \frac{1}{k_s - 1} \leq 1$. By homogeneity, we assume they have identical constraints on the bounds of load and storage spaces. We apply selective replication, instead of full and excess replication. The offline version of this algorithm is in [7]. Unlike the first algorithm, the term S in the time complexity is not needed here. This is because we do not move a document from one server to another, but de-replicate some documents by overwritting their replicas. This algorithm beats the first one when $(k_l - \frac{1}{2}) + k_s < 6$. The benefit in the bounds of load and storage space is the return for handling extra replicas. The freedom of re-allocation supports this online version.

INFOSCALE '06. Proceedings of the First International Conference on Scalable Information Systems, May 29-June 1 2006, Hong Kong
© 2006 ACM 1-59593-428-6/06/05...$5.00

Many papers have addressed the load balancing problem in distributed file server systems [1, 2, 3, 5, 6, 7, 8], but none of them considered the case for deletion. In general, what we can do for deletion is less than insertion because deletion will disturb some well-developed properties. The contribution of our third algorithm is to narrow this gap, and balance the load and storage space with performance slightly worse than document placement.

For any input sequence of any combination of insertions and deletions, the algorithm will bound the load and storage space by $(k_l + \epsilon)L$ and $(k_s + \epsilon)S$, respectively. The time needed is $O(S + \frac{1}{\epsilon}\log M)$ for each deletion. Note that one can immediately come up with an example of impossibility if we have no freedom to re-allocate the documents. Even with this freedom, the price for compensating the disturbance by deletion is an additive term "$+\epsilon$" in the first factor of the bounds for the load and storage space, compared with the pure placement case. Document deletion is frequent in file servers of temporary documents. We can apply it to highly volatile file systems. Lastly, we will show insertion of a new empty server is a special case of document deletion in the algorithmic context of load balancing. In other words, using the last algorithm, insertion of an empty server will affect little on the balancing of load and storage space, and we can scale up the number of servers online without pay much cost.

Due to the page length limit, we skip the correctness proof and time complexity analysis of the second and third algorithms. One could refer to our full paper.

2 Definitions and Models

Each document has two fundamental attributes, namely load and size. There are M servers and N documents. The value of N changes upon each placement and deletion. The ith document has positive load l_i and size s_i, $\forall i \in [1, N]$. The load and storage space of a server is the summation of loads and sizes of all documents stored, respectively. For all $j \in [1, M]$, the load of the jth server is denoted as $\mathcal{L}_j$ and the storage space $\mathcal{S}_j$. We do not assume any fixed limit on their values; however, there is still a need to balance them among the servers.

Let $\overline{L}$ and $\overline{S}$ be the average load and storage space. Therefore, we have $\overline{L} = \frac{\sum_{i\in[1,N]} l_i}{M}$, and $\overline{S} = \frac{\sum_{i\in[1,N]} s_i}{M}$. Let L and S be the optimal bounds on the load and storage space of each server, respectively, where $L = \max(\max_{i\in[1,N]} l_i, \overline{L})$ and $S = \max(\max_{i\in[1,N]} s_i, \overline{S})$. $\forall j \in [1, M]$, let $p_l^j > 2$ and $p_s^j > 2$ be two numbers satisfying

$$p_l^j + p_s^j \geq 6. \tag{1}$$

These values reflect the tradeoff between the bounds of the load and storage space for the servers in the first two algorithms. That is, $\forall j \in [1, M]$, $\mathcal{L}_j < p_l^j L$ and $\mathcal{S}_j < p_s^j S$ in the first algorithm. The second and final results are for homogeneous servers. Let $k_l > 2$ and $k_s > 2$ be two numbers satisfying

$$\frac{1}{k_l - 1} + \frac{1}{k_s - 1} \leq 1. \tag{2}$$

Note that we define k_l and k_s in the same way as in [7]. Our second result is that $\mathcal{L}_j < (k_l - \frac{1}{2})L$ and $\mathcal{S}_j < k_s S$, and the final result $\mathcal{L}_j < (k_l + \epsilon)L$ and $\mathcal{S}_j < (k_s + \epsilon)S$, $\forall j \in [1, M], \forall 0 < \epsilon \leq 1$.

The heterogeneity among servers are that each server can have individual bounds on load and storage space which satisfy the Inequality (1), while homogeneity are that all servers have the same bounds. Note that whenever the value of N changes, the values of L, S, $\overline{L}$, $\overline{S}$, $\mathcal{L}_j$ and $\mathcal{S}_j$ change. However, the values of k_l, k_s, p_l^j and p_s^j are the initial parameters of the server systems, and therefore, stable until the next system maintenance. Since these four values are directly related to the balancing of servers' performance, it is reasonable to assume they are constants.

We apply the B^0-tree used in [7] for storing the information of the servers. This data structure, which stores a set $T = \{(x, y) | x, y \in R^+\}$ of size M, is widely employed throughout this paper. We assume the elements in T are unique [1]. We let $\mathcal{A}$ be the algorithm in [7] for performing searching and updating on a B^0-tree. For any input $X \in R^+$, $\mathcal{A}$ can search an element (x, y) in T and perform updating within $O(\log M)$ time, where y is the smallest possible value such that $x < X$. In other words, for any element (x', y') in T, $[y' < y] \Rightarrow [x' \geq x]$. In the case that $x \geq X$ for each $(x, y) \in T$, $\mathcal{A}$ will output a false. We will not discuss $\mathcal{A}$ and B^0-tree in the paper, the readers may refer to our work in [7].

By using $\mathcal{A}$, we can easily construct an algorithm $\mathcal{A}'$ which can search an element $(x, y) \in T$ such that $x < X$ and $y < Y$, for any input (X, Y) where $X, Y \in R^+$, and update x and y if needed. $\mathcal{A}'$ will output a false if searching is not successful. We can construct another algorithm $\mathcal{A}''$ to output (x, y) for $x \leq X$ and $y \leq Y$ by running $\mathcal{A}'$ on input $(X + \delta, Y + \delta)$ such that $X + \delta$ and $Y + \delta$ are greater than X and Y, and less than any elements greater than X and Y, if any, respectively. δ can be found in $O(\log M)$ time using some common data structures [4].

For simplicity, we skip the discussion of some necessary but trivial steps for updating and maintaining the data structures used.

3 Placement for Heterogeneous Servers

Our aim is to bound the load and storage space of the jth server by $p_l^j L$ and $p_s^j S$, respectively, $j \in [1, M]$. When a

[1]Precisely, we can define $T = (B_1, B_2, \ldots, B_M)$, where $B_i = (x, y)$ for some $x, y \in R^+$, $\forall i \in [1, M]$

new document comes, we first update L and S. If there exists an $j \in [1, M]$ such that $\mathcal{L}_j < (p_l^j - 1)L$ and $\mathcal{S}_j < (p_s^j - 1)S$, we place the new document into this jth server and the load and storage space of the server are bounded by $p_l^j L$ and $p_s^j S$, respectively. Then, it is done. For implementation, we store $(\frac{\mathcal{L}_j}{p_l^j - 1}, \frac{\mathcal{S}_j}{p_s^j - 1})$ in a B^0-tree, $\forall j \in [1, M]$, and then apply $\mathcal{A}'$ on it with input (L, S). Therefore, it takes $O(\log M)$ time.

If no such server exists, we have

$$\mathcal{L}_j \geq (p_l^j - 1)L \quad \text{or} \quad \mathcal{S}_j \geq (p_s^j - 1)S, \qquad (3)$$

for all $j \in [1, M]$. In this case, at least one of the following cases must be true.

(A) $\exists j \in [1, M]$ such that $\mathcal{L}_j \leq L$, $\mathcal{S}_j \geq (p_s^j - 1)S$ and $p_s^j < 3$.

(B) $\exists k \in [1, M]$ such that $\mathcal{S}_k \leq S$, $\mathcal{L}_k \geq (p_l^k - 1)L$ and $p_l^k < 3$.

Reason: We define the capacity index C_j for the jth server to be $\frac{\mathcal{L}_j}{L} + \frac{\mathcal{S}_j}{S}$. Obviously, $\sum_{j \in [1,M]} C_j = 2M$. Recall that $p_l^j, p_s^j > 2$, for all $j \in [1, M]$. If both of the above cases are false, $[\mathcal{L}_j \geq (p_l^j - 1)L] \Rightarrow [\mathcal{S}_j > S$ or $p_l^j \geq 3]$, and $[\mathcal{S}_j \geq (p_s^j - 1)S] \Rightarrow [\mathcal{L}_j > L$ or $p_s^j \geq 3]$. By (3), we have $C_j > 2$. A contradiction. $\Diamond$

If both of the above cases are true, we simply swap the contents of the two corresponding servers, and therefore, any one of the resulting servers can be used to place the new document. Consider the time complexity. We first store a set $\{(\mathcal{L}_j, \mathcal{S}_j) | p_s^j < 3, j \in [1, M]\}$ in a B^0-tree and apply algorithm $\mathcal{A}$ with input L. It will return the server for case (A). For getting a case (B) server, we store a set $\{(\mathcal{S}_j, \mathcal{L}_j) | p_l^j < 3, j \in [1, M]\}$ in another B^0-tree and apply algorithm $\mathcal{A}$ with input S. Totally, it takes $O(\log M)$ time for searching for these two servers and $O(S)$ time for swapping, and $O(\log M)$ time for updating. Hence, time needed is $O(S + \log M)$.

If only one of the above cases is true, without loss of generality, we assume case (A) is true, and case (B) is false. Therefore, $\forall j \in [1, M]$,

$$[\mathcal{L}_j \geq (p_l^j - 1)L] \Rightarrow [\mathcal{S}_j > S \text{ or } p_l^j \geq 3]. \qquad (4)$$

We now divide the whole set of servers into different types. A type-1 server is a server who meets the condition of case (A). $\forall j \in [1, M]$, if $\mathcal{L}_j \leq L$, $\mathcal{S}_j \geq (p_s^j - 1)S$ and $p_s^j \geq 3$, the jth server is called type-2 server. It is called type-3 server if $\mathcal{L}_j > L$ and $\mathcal{S}_j \geq (p_s^j - 1)S$. It is called type-4 server if $\mathcal{L}_j \geq (p_l^j - 1)L$ and $S < \mathcal{S}_j < (p_s^j - 1)S$. It is called type-5 server if $\mathcal{L}_j \geq (p_l^j - 1)L$, $\mathcal{S}_j \leq S$, and $p_l^j \geq 3$. The disjointness of the above types is clear. For completeness, we can easily check by using (3) and (4).

Among all type-1 and type-5 servers, we search the ith and kth servers, say, with smallest possible storage spaces, respectively, where $i, k \in [1, M]$. The existence of a type-1 server is a direct consequence of the above case (A). We now argue that a type-5 server with storage space less than S exists. Since case (A) is true, there exists a server with storage space greater than S. Then, there must be another server, which we call the kth server, with storage space less than S; otherwise, the storage space of the whole server system will be greater than MS. By (3), we have $\mathcal{L}_k \geq (p_l^k - 1)L$, which implies $p_l^k \geq 3$ by (4). Hence, this server is a type-5 server with storage space less than S. Let $\mathcal{S}_k = \delta S$, where $0 < \delta < 1$.

Initially, both servers cannot accept the new document because the storage space of the ith server and the load of the kth server are approaching to their limits. However, if we take away a minimal subset Δ of documents from the kth server such that the total load in Δ is at least L, it can have a room for the new document. Since the set Δ is not necessarily optimal [2], linear time $O(S)$ is enough for constructing it. Obviously, the total load and size in Δ is less than $2L$ and $\mathcal{S}_k$, respectively. We put the documents of Δ into the ith server, and we need to prove the new $\mathcal{L}_i$ and $\mathcal{S}_i$ are less than $p_l^i L$ and $p_s^i S$, respectively.

First, since $p_s^i < 3$, by (1), we have $p_l^i > 3$. The new load of the ith server is less than $L + 2L = 3L < p_l^i L$. For storage space, assume the contrary that the new storage space is at least $p_s^i S$. Therefore, $\mathcal{S}_i \geq (p_s^i - \delta)S > (2 - \delta)S$. Assume there are n_t type-t servers, $t \in [1, 5]$. Then, $\sum_{t \in [1,5]} n_t = M$. Consider the total load, we have

$$2n_5 + n_4 + n_3 < M. \qquad (5)$$

The total storage space is at least $(n_1(2 - \delta) + 2n_2 + n_3 + n_4 + n_5\delta)S$. Rearranging the terms, it becomes $(M + n_2)S + (M - n_2 - n_3 - n_4 - 2n_5)(1 - \delta)S > MS + (M - n_3 - n_4 - 2n_5)(1 - \delta)S$. By (5), it is greater than MS. A contradiction. Hence, the new storage space of the ith server is less than $p_s^i S$.

To find the ith server is the same as to find a case (A) server. For the kth server, we store a set $\{(\mathcal{L}_j, \mathcal{S}_j) | p_l^j \geq 3, j \in [1, M]\}$ in a B^0-tree and apply algorithm $\mathcal{A}$ with input ∞. That means, we need not check $(p_l^k - 1)L \leq \mathcal{L}_j < p_l^k L$ because it is implied. We only target at a minimum $\mathcal{S}_k$ and $p_l^k \geq 3$ only. The total time needed for placement is then bounded by $O(S + \log M)$.

4 Placement with Replication for Homogeneous Servers

Our aim is to bound the load and storage space of the jth server by $(k_l - \frac{1}{2})L$ and $k_s S$, respectively, $j \in [1, M]$.

[2]The problem is NP-complete for optimal solution.

For each document, there is at most one replica, that is, two copies in total. If a document has no replica, it is a *single document*. If a document has two copies, both of the original document and its replica are called *twins*. Each of them shares half of the load. Obviously, it is useless to place them in the same server. For the size, both twins will have the whole size without any reduction. Thus, in view of the whole system, the document's load has no change in total, but the storage required is double. In other words, replication should be done carefully in order to make its advantage significant. In an online environment, any necessary replicas at a time may become redundant later. That means, we need to create and remove twins from time to time. In the case that a twin is removed, its other twin will take up all the load and become a single document. Note that we will not remove both twins of a document; otherwise, the document will be deleted from the system. Each server is allowed to store at most one twin. Therefore, there are at most M twins from $\lfloor \frac{M}{2} \rfloor$ documents. To guarantee that we do not perform any unnecessary removal of twins, we say a twin is redundant if the load of the server containing it is less than $(k_l - 1)L$. Along with it, if one of the twins of a document is redundant, we can remove its another twin so that the redundant twin will become a single document with full load, and the server containing it will have a load less than $(k_l - 1)L + \frac{L}{2} = (k_l - \frac{L}{2})L$, and the storage space is kept unchanged.

Define two sets,

$D = \{(\mathcal{S}_j, \mathcal{L}_j) |$ the jth server has 1 twin, $j \in [1, M]\}$,
and
$E = \{(\mathcal{S}_j, \mathcal{L}_j) |$ the jth server has 0 twin, $j \in [1, M]\}$.

We store them in two B^0-trees, respectively, so that we can apply algorithms $\mathcal{A}$ and $\mathcal{A}'$ to them later. For simplicity, we define the $MOVE$ procedure as follows:

```
Procedure $MOVE(U, V; \mathcal{S}_i, \mathcal{L}_i; s, l)$;
// $\{U, V\} = \{D, E\}$, and $s$, $l$ can be $\pm$ve.
        Remove $(\mathcal{S}_i, \mathcal{L}_i)$ from set $U$;
        $\mathcal{S}_i := \mathcal{S}_i + s$;
        $\mathcal{L}_i := \mathcal{L}_i + l$;
        Store $(\mathcal{S}_i, \mathcal{L}_i)$ into set $V$;
```

The placement algorithm is shown below.

```
On receiving a document with load $l$ and size $s$
1.      Update $L$ and $S$;
2.      Do twice
2.1         Run $\mathcal{A}'$ on $D$ on input $(\infty, (k_l - 1)L)$;
2.2         If output is $(\mathcal{S}_k, \mathcal{L}_k)$,    //not a false
2.2.1           Suppose the twin in the $k$th server
                has load $l'$ and storage space $s'$,
                and the $i$th server contains its
                another twin;
2.2.2           Run $MOVE(D, E; \mathcal{S}_i, \mathcal{L}_i; -s', -l')$;
2.2.3           Run $MOVE(D, E; \mathcal{S}_k, \mathcal{L}_k; 0, l')$;
3       Run $\mathcal{A}$ on $E$ with input $(k_l - 1)S$ and
        output $(\mathcal{S}_k, \mathcal{L}_k)$;
4       Place the document into the $k$th server;
5       If $\mathcal{L}_k + l \geq (k_l - \frac{1}{2})L$
5.1         Then
5.1.1           Run $MOVE(E, D; \mathcal{S}_k, \mathcal{L}_k; s, \frac{l}{2})$;
5.1.2           Run $\mathcal{A}$ on $E$ with input $(k_l - 1)S$
                and output $(\mathcal{S}_i, \mathcal{L}_i)$;
5.1.3           Place the document as a twin into
                the $i$th server;
5.1.4           Run $MOVE(E, D; \mathcal{S}_i, \mathcal{L}_i; s, \frac{l}{2})$;
5.2         Else
5.2.1           Update $\mathcal{L}_k$ by $\mathcal{L}_k + l$
                and $\mathcal{S}_k$ by $\mathcal{S}_k + s$;
```

5 Placement and Deletion for Homogeneous Servers

In this section, we consider the input to be a sequence of any combinations of placements or deletions of documents. We will give an algorithm which bounds the load and storage space of the jth server by $(k_l + 1)L$ and $(k_s + 1)S$, respectively, after each operation. In Theorem 1, we will state the generalized result.

Except the server whose document has just deleted, each server will experience a relative "increase" of load and storage space. When a "load-giant" or "size-giant" is deleted, some servers may lose a shelter, and their loads or sizes may rocket. Precisely, deletion of a document may lead to a vast drop of L and S, which may be followed by a violation of the load and storage space bounds $(k_l + 1)L$ and $(k_s + 1)S$ for many servers. To avoid the problem, document re-allocation will be performed. For the online purpose, the time taken is bounded by $O(S + \log M)$. Therefore, we cannot re-allocate many documents at a time. We re-allocate the documents in the most potential overflowing servers. The second strategy is to isolate those documents with extremely high load or large size, in order to avoid any insertion of documents on their tops. Note that the load/size of a document can be extremely high at a time and become normal when the average load/size surges upon the arrival of new documents. On the other hand, a normal document can become a "load/size-giant" after a number of document deletions. So, those isolated documents may not always be isolated. We use some dynamic data structure for handling the isolation. Since the shock to L and S, and even $\overline{L}$ and $\overline{S}$, from each placement and deletion of a "giant" is inevitable, our third strategy is to introduce another parameters, namely λ and σ as defined below. We will see the effect on these parameters is reduced to a manageable level. The price for these parameters is $O(\log M)$ time complexity for updating and maintaining the necessary data structures while updating L, S, $\overline{L}$ and $\overline{S}$ requires only $O(1)$ time.

As load and storage space are symmetric in this section, for conciseness, we will skip some discussion for storage space as long as no ambiguity exists.

Define λ to be the $\lceil \frac{M}{k_l-1} \rceil$th largest load, and σ the $\lceil \frac{M}{k_s-1} \rceil$th largest storage space among all servers. Although their values can vary upon placement and deletion, they are less affected by giants compared with L, S, $\overline{L}$ and $\overline{S}$. Let l_j^{last} be the load of the last input document in the jth server, and s_j^{last} the size of that. Define the sets

$$\begin{aligned} F &= \{(\mathcal{S}_j, \mathcal{L}_j) | j \in [1, M]\}, \\ G &= \{(\mathcal{S}_j, \mathcal{L}_j) \in F | \text{the } j\text{th server} \\ &\qquad \text{has more than one document}\}, \end{aligned}$$

Below we give two procedures, namely $Find$, $Vacate()$, and $Mark()$.

```
Procedure $Find$:
1.      Run $\mathcal{A}''$ on $F$ with input $(\sigma, \lambda)$, and get
        output $(\hat{\mathcal{S}}_j, \hat{\mathcal{L}}_j)$;
2.      Return $j$;
```

```
Procedure $Vacate(j)$:
1.      Let $\lambda' := \frac{\lambda}{k_l}$; $\sigma' := \frac{\sigma}{k_s}$;
2.      Take away all documents from the $j$th
        server and put them into buckets such
        that (1) There is at most one bucket
        having total load at most $\frac{\lambda'}{2}$ and total
        size $\frac{\sigma'}{2}$, and (2) if a bucket has more
        than one document, the total load lies
        between $\frac{\lambda'}{2}$ and $\lambda'$, or total size
        between $\frac{\sigma'}{2}$ and $\sigma'$;
3.      Return the set of buckets;
```

```
Procedure $Mark(j, \lambda)$:
1.      Remove all the existing symbols of the
        $j$th server;
2.      If $\mathcal{L}_j < \frac{k_l+1}{k_l}\lambda$
2.1         Mark the $j$th server with a symbol $\Lambda$;
3.      If $\mathcal{S}_j < \frac{k_s+1}{k_s}\sigma$
3.1         Mark the $j$th server with a symbol $\Xi$;
```

Define

$$\begin{aligned} G_\Lambda &= \{(\mathcal{S}_j, \mathcal{L}_j) \in G | \text{the } j\text{th server marked by } \Lambda\}, \\ G_\Xi &= \{(\mathcal{S}_j, \mathcal{L}_j) \in G | \text{the } j\text{th server marked by } \Xi\}. \end{aligned}$$

We now give Algorithms $Placement$ and $Deletion$.

```
Algorithm $Placement$:
On receiving a document with load $l$ and size $s$
1.        Place the document into an empty server,
          if any;
2.        If $l < \lambda$ and $s < \sigma$
2.1           Then
2.1.1             Run $Find$ and get output $j$;
2.1.2             Place the input document into the
                  $j$th server;
2.1.3             Run $Mark(j, \lambda)$;
2.1.4             Update $\lambda$ and $\sigma$;
2.2           Else
2.2.1             Run $\mathcal{A}'$ on $F$ with input $(\sigma, \lambda)$
                  and get output $(\mathcal{S}_j, \mathcal{L}_j)$;
2.2.2             Run $Vacate(j)$ and get output $X$;
2.2.3             Remove all symbols of the $j$th server;
2.2.4             Place the input document into the $j$th
                  server;
2.2.5             Update $\lambda$ and $\sigma$;
2.2.6             For each bucket $B$ in $X$
2.2.6.1               Run $Find$ and get output $j$;
2.2.6.2               Place all documents in $B$ into the
                      $j$th server;
2.2.6.3               Run $Mark(j, \lambda)$;
2.2.6.4               Update $\lambda$ and $\sigma$;
```

```
Algorithm $Deletion$:
On deleting a document from the $k$th server
1.        Update $\lambda$ and $\sigma$;
2.        Remove all symbols of the $k$th server if it is
          empty;
3.        If $G \neq \emptyset$
3.1           Do $2k_l$ times
3.1.1             Pick a $j$ such that $(\mathcal{S}_j, \mathcal{L}_j) \in G_\Lambda$ and
                  $\forall(\mathcal{S}_i, \mathcal{L}_i) \in G_\Lambda$, $\mathcal{L}_i \leq \mathcal{L}_j$;
3.1.2             Run $Vacate(j)$ and get output $X$;
3.1.3             Remove all symbols of the $j$th server;
3.1.4             Update $\lambda$ and $\sigma$ as if the documents
                  in the buckets of $X$ have been
                  expelled;
3.1.5             For each bucket $B$ in $X$
3.1.5.1               Run $Placement$ with $B$ as a new
                      document input;
3.1.6             Pick a $j$ such that $(\mathcal{S}_j, \mathcal{L}_j) \in G - G_\Lambda$
                  and $\forall(\mathcal{S}_i, \mathcal{L}_i) \in G - G_\Lambda$,
                  $\mathcal{L}_i - l_i^{last} \leq \mathcal{L}_j - l_j^{last}$;
3.1.7–3.1.10      Repeat Steps 3.1.2–3.1.5;
3.2           Do $2k_s$ times
3.2.1             Pick a $j$ such that $(\mathcal{S}_j, \mathcal{L}_j) \in G_\Xi$ and
                  $\forall(\mathcal{S}_i, \mathcal{L}_i) \in G_\Xi$, $\mathcal{S}_i \leq \mathcal{S}_j$;
3.2.2–3.2.10      Similar to Steps 3.1.2-3.1.10.
```

Theorem 1 states our result and Theorem 2 the extension.

Theorem 1 $\forall 0 < \epsilon \leq 1$, *if* $M \geq \max(\frac{4}{\epsilon}(k_l - 1)^2, \frac{4}{\epsilon}(k_s - 1)^2)$*, there exists an algorithm to accept a sequence of any combinations of placements and deletions such that after each* $O(S + \frac{1}{\epsilon}\log M)$*-time placement or deletion, the load of each server is less than* $(k_l + \epsilon)L$*, and storage space* $(k_s + \epsilon)S$*.*

Theorem 2 $\forall 0 < \epsilon \leq 1$, *if* $M \geq \max(\frac{4}{\epsilon}(k_l - 1)^2, \frac{4}{\epsilon}(k_s - 1)^2)$*, there exists an* $O(S+\frac{1}{\epsilon}\log M)$*-time algorithm for balancing the loads and storage spaces on each server insertion such that the load of each server is less than* $(k_l + \epsilon)L$*, and storage space* $(k_s + \epsilon)S$*.*

6 Conclusion

In this paper, we give three results for balancing the loads and storage spaces among server.

In practice, in order to achieve the best bounds, we need to equalize both sides of the inequalities (1) and (2). Then the system designer can determine the tradeoff between load and storage space. Further research can focus on the online varying k_l and k_s, and p_l^j and p_s^j, $j \in [1, M]$, as long as k_l and k_s are bounded by a constant. The values of p_l^j and p_s^j are between 2 and 4 by definition.

The practicability of the results in this paper is first based on the online property of all there algorithms. By online, we mean the time cost for each operation is reasonably bounded. One example is new server insertion. The users do not need to suffer from system suspension, and the system does not suffer from an imbalance of load or storage space. Second, heterogeneity is obviously an advantage for scalability. Last, document deletion is frequent in file servers of temporary documents. A common belief is that deletion can disturb the existing well-balance condition. The constribution of our third algorithm turns down this belief, and balances the load and storage space with performance slightly worse than document placement. We can apply it to highly volatile file systems. Further research may be done to combine heterogeneity, replication and deletion in an online algorithm.

In the case that the documents are usually of small load or size, the performance must be much better than our upper bounds and the solution to such systems must be simpler. However, unless all special cases are filtered out, a proven upper bounded is still needed. The upper bounds in this paper is actually a guarantee of the practical performance.

References

[1] R.B. Bunt, D.L. Eager, G.M. Oster and C.L. Williamson, “Achieving Load Balance and Effective Caching in Clustered Web Servers”, *Proc. of the 4th International Web Caching Workshop*, San Diego, CA, USA, March, 1999.

[2] L.C. Chen and H.A. Choi, “Approximation Algorithms for Data Distribution with Load Balancing of Web Servers”, *Proc. of IEEE International Conference on Cluster Computing*, 274–281, Newport Beach, CA, USA, October, 2001.

[3] M. Colajanni, P.S. Yu, V. Cardellini, “Dynamic Load Balancing in Geographically Distributed Heterogeneous Web Servers”, *Proc. of International Conference on Distributed Computing Systems*, 295–302, Amsterdam, Netherlands, May, 1998.

[4] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, “Introduction to Algorithms”, McGraw-Hill, New York, 2nd Edition, 2001.

[5] E.D. Katz, M. Butler and R. McGrath, “A Scalable HTTP Server: The NCSA Prototype”, *Computer Networks and ISDN Systems*, vol. 27, No. 2, 155–164, 1994.

[6] B. Narendran, S. Rangarajan and S. Yajnik, “Data Distribution Algorithms for Load Balanced Fault-Tolerant Web Access”, *Proc. of the 16th Symposium on Reliable Distributed Systems*, 97–106, Durham, NC, USA, October, 1997.

[7] S.S.H. Tse, “Approximation Algorithms for Document Placement in Distributed Web Servers”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, No. 6, 489–496, June 2005.

[8] L. Zhuo, C.L. Wang and F.C.M. Lau, “Load Balancing in Distributed Web Server Systems with Partial Document Replication”, *Proceeding of the 2002 International Conference on Parallel Processing*, 305–312, Vancouver, Canada, August 2002.