ETH zürich

ETH Library

Distributed file organization with scalable cost/performance

Report

Author(s):
Vingralek, Radek

Publication date:
1993

Permanent link:
https://doi.org/10.3929/ethz-a-000923052

Rights / license:
In Copyright - Non-Commercial Use Permitted

Originally published in:
ETH, Eidgenössische Technische Hochschule Zürich, Departement Informatik, Institut für Informationssysteme 207

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

Eidgenössische
Technische Hochschule
Zürich

Departement Informatik
Institut für
Informationssysteme

Radek Vingralek
Yuri Breitbart
Gerhard Weikum
Raj Yavatkar

Distributed File Organization with Scalable Cost/Performance

Eidg. Techn. Hochschule Zürich
Informatikbibliothek
ETH-Zentrum
CH-8092 Zürich

December 1993

207

ETH Zürich
Departement Informatik
Institut für Informationssysteme
Prof. Dr. H.-J. Schek

This research has been undertaken while two of the authors, Yuri Breitbart and Radek Vingralek, were visiting the database research group at ETH Zurich for 12 and 8 months, respectively.

The work of these two authors was sponsored in part by a grant from Hewlett-Packard Corporation and by the U.S. National Science Foundation under grant IRI-92121301.

Author's addresses:

R. Vingralek, Y. Breitbart, R. Yavatkar, Department of Computer Science, University of Kentucky, Lexington, KY 40506, USA

G. Weikum, Institut für Informationssysteme, ETH Zentrum, CH-8092 Zurich, Switzer-land. e-mail: weikum@inf.ethz.ch

© 1993 Departement Informatik, ETH Zürich

Distributed File Organization with Scalable Cost/Performance

Radek Vingralek[1] Yuri Breitbart[1] Gerhard Weikum
Raj Yavatkar[2]

Abstract

This paper presents a distributed file organization for record-structured, disk-resident files with key-based exact-match access. The file is organized into buckets that are spread across multiple servers, where a server may hold multiple buckets. Client requests are serviced by mapping keys onto buckets and looking up the corresponding server in an address table. Dynamic growth in terms of file size and access load is supported by bucket splits and bucket migration onto other existing or newly acquired servers.

The major challenge that we are addressing is to achieve scalability in the sense that both the file size and the access load (i.e., throughput of client requests) can be scaled up by linearly increasing the number of servers and dynamically redistributing data. Unlike previous work with similar objectives, our data redistribution considers explicitly the cost/performance ratio of the system by aiming to minimize the number of servers that are acquired to provide the required performance. A new server is acquired only if the overall server utilization in the system does not drop below a specified threshold. Preliminary simulation results show that the goal of scalability with controlled cost/performance is indeed achieved to a large extent.

[1]On leave from the Department of Computer Science, University of Kentucky, Lexington, KY 40506, U.S.A.
[2]Department of Computer Science, University of Kentucky, Lexington, KY 40506, U.S.A.

1 Introduction

1.1 Problem Statement

Data-intensive computer applications are posing ever-increasing demands in terms of storage and performance capacity. One cost-effective approach to meeting such requirements is to exploit distributed storage and computing resources in a client-server architecture. Recently, a number of proposals have been made for organizing record-structured, key-accessed, disk-resident files so that the file size and the file access rate can be increased in a theoretically unlimited way by dynamically distributing the file across a number of servers (e.g., [MS91, Dev93, JK93, LNS93]).

Most notably, Litwin, Neimat, and Schneider [LNS93] have developed a distributed version of linear hashing, coined LH*, which assigns hash buckets to servers and adds servers dynamically upon bucket splits. The clients locate servers for insertion, update, deletion, and key-based retrieval through a hash function whose target domain is dynamically expanded as buckets are split and servers are added, based on the principle of linear hashing. To limit the communication overhead, clients learn about bucket splits and the resulting changes to the server-locating hash function only when a client request is directed to the "wrong" server. In such a case, the server forwards the client's request to the proper server, and notifies the client in order to update the client's stale information on the hash function. The most salient feature of this approach is that its communication overhead is largely independent of the number of servers and clients in the system; thus, it is considered a scalable approach.

More precisely, the objective of *scalability* means the following [Gr91, DG92]. Starting with a system where one server manages a file of a specific size that is accessed by a specific number of clients at a specific rate, a scalable approach can efficiently manage a file that is n times bigger and accessed by n times more clients at the same per-client rate, by adding servers and distributing the file across these servers. That is, the system throughput can be increased by a factor of n, and, furthermore, the response time of the clients' requests should be as good as in the one-server case. As perfect scalability (for non-trivial workloads) is achievable only theoretically, we usually speak of a scalable approach already if response time is nearly constant (e.g., within a factor of 2) for reasonably large values of n (e.g., $n = 100$) and increases only very slowly for very large values of n.

What is missing in this discussion of scalability, however, is a consideration of the cost that is incurred by increasing the storage and performance capacity. Ideal scalability would require that a capacity gain of n is achieved by increasing the number of servers also by a factor of n [DG92]. However, the bucket splitting mechanism of LH* and

similar work inevitably increases the number of servers by a factor higher than n. Expanding a file that exhausts the complete capacity of one server by a factor of 100 may, for example, lead to a system of 140 servers with an average utilization of 70%, where utilization includes both storage and performance capacity. One may argue that such an approach is still truly scalable as the resulting 140 servers are actually used only partially, with their remaining capacity being available to other applications. In this paper, we take a different viewpoint, however. In our cost consideration, we would include all 140 servers for the following reasons:

- The servers need some free capacity to support bucket growth before reaching the splitting point. This extra capacity is not truly available to other applications since it is withdrawn dynamically as the bucket grows. Thus, other applications cannot really count on this "free" capacity.
- More servers imply a higher cost in terms of system administration (e.g., file backups) and availability (e.g., a higher failure rate), regardless of how well loaded the servers are. As it is hard to quantify these additional costs, we assume throughout this paper that the cost of the system is proportional to the number of servers that are involved in the management of the file.

1.2 Our Approach

This paper presents a new approach for hash-based distributed files that strives for the objective of "cost-conscious" scalability, by aiming to minimize the number of servers that are used for providing the necessary performance capacity. Thus, our approach has a built-in *control of cost/performance*. Our approach is based on the following key concepts:

1. In contrast to LH*, we assume an indirection between bucket numbers and server numbers, so that multiple buckets can be assigned to the same server. This indirection is implemented by an *address table*, similar to the directory of the extendible hashing method [FNPS79] but stored in a more compact way. Similarly to LH*, updates to the address table are propagated to clients and other servers in a deferred manner; forwarding is used to locate buckets when a client has a stale address table.
2. For each server, we employ a *local load control* that prevents a server from being overloaded, by redistributing buckets upon reaching a specified server utilization. Having an explicit mapping of buckets onto servers gives us additional flexibility to redistribute the load of an overloaded server: we can either *split* one or more buckets of the server or we can *migrate* a bucket to another server,

and, orthogonally to this issue, we can acquire a new server for the buckets to be moved or we can merely move the buckets to an existing server.

3. Hashing is used merely to distribute keys across buckets. The internal organization of buckets can be chosen freely, and may vary between servers. For example, a single bucket could be organized locally by linear hashing or as a B^+-tree. For the scope of this paper, the only important point is that a bucket constitutes a certain access load that has to be sustained by the server on which the bucket resides. (A bucket also constitutes a certain storage cost, but this is of minor relevance in this paper.)

4. The decisions about splitting versus migrating a bucket and acquiring a new server versus redistributing across existing servers depend on the overall utilization of (the performance capacity of) the existing servers. To make an intelligent decision, our approach makes an "educated guess" of the total load of the entire system (i.e., all servers), based on a probabilistic model with sampling-based information on bucket sizes. For ease of presentation, we assign this estimation task to a logically centralized process that we refer to as the *file advisor*; however, our general approach would allow a distributed estimation as well.

5. The guideline for the decisions mentioned under point 4 is the following. As long as the estimated total system utilization is or would fall below some threshold, we do not acquire a new server and rather prefer a redistribution of buckets among the existing servers. This policy aims to keep the number of servers as small as possible, thus ensuring a good cost/performance ratio and aiming at "cost-conscious" scalability (in the sense of Section 1.1).

It is important to note that our main concern is the control and (balanced) distribution of the access load rather than the avoidance of overflow chains as in conventional dynamic hashing methods (see, e.g., [ED88, Lar88]). We assume, for simplicity, that the access load of a bucket is proportional to its size (i.e., the number of records in the bucket). This is a reasonable assumption for hash buckets with a sufficiently large number of different keys, even though it disregards the possibility of skew values [WDJ91] (which may occur especially if the search key of the file allows duplicates). The *load of a server* is the accumulated load of its buckets, and the *total system load* is the accumulated load of the servers.

These notions of server and system load may appear to coincide with the classical notion of the "load factor" of a hash file, applied to the buckets of one server or to all buckets in the system, respectively. However, the difference is that the load factor reflects the storage utilization rather than the utilization of performance capacity (i.e., the percentage of the maximum throughput that can be sustained).

In the rest of the paper we concentrate solely on the issue of performance utilization and disregard storage utilization. We assume that each server has the same performance capacity (expressed in the maximum amount of data that the server can hold but actually reflecting the access rate to this data). In principle, our approach could allow servers with different performance capacities (e.g., a heterogeneous workstation farm), but the details of this generalization are left for future work.

1.3 Related Work

Our approach is most closely related to the work of [LNS93] and [Dev93]. In particular, we have adopted the idea of using an address table from [Dev93], to gain flexibility in the placement and splitting order of buckets. However, none of this previous work considers the cost of scaling up a distributed system explicitly, and thus no attempt is made to minimize the number of servers across which a file is spread. Note that the consideration of the load factor of a hash file is a storage utilization issue, whereas we assume that storage capacity is relatively uncritical and are rather concerned with the control of cost/performance. Thus, the notion of an *overflown bucket* differs fundamentally from the notion of an *overloaded server.*

Our approach draws on the option of redistributing data by means of bucket migration. Data migration as a means for load balancing of data management systems has been considered in a number of papers on multi-disk and distributed file systems (e.g., [WSZ91, WJ92, HW94]). However, the underlying file model of these approaches is that of Unix files (i.e., bytestrings), whereas we assume record-structured files with key-based access. To the best of our knowledge, none of the previous approaches to distributed record-structured file organizations has considered the migration of file portions to avoid or defer split-like reorganizations.

Finally, our approach has adopted some of the principles that have been developed in the context of distributed CPU load sharing and process migration (e.g., [BS85, ELZ86, LLM88] to mention some of the seminal work in this area). In particular, our approach to estimating the total system load resembles some of the heuristics that are used in adaptive load sharing.

1.4 Overview of the Paper

The rest of the paper is organized as follows. Section 2 gives an overview of our distributed file algorithm. Then, Sections 3 and 4 present details of file advisor methods for tracking the load of servers and for estimating the total system load

in order to make decisions about when a new server should be acquired. Section 5 presents preliminary results of a simulation study that we have been conducting. We conclude with a brief discussion of possible extensions to and generalizations of our approach.

2 Overview of the Distributed File Algorithm

2.1 System Model

Consider a file consisting of keyed records that is spread across a number of servers connected by a network. A file is stored as a collection of n buckets distributed among m servers ($m \leq n$). A server keeps records of the file primarily on disk and we assume that servers are dedicated to managing a distributed file. We assume that there is a distributed group of clients that issue requests to insert, delete, and retrieve records for a given key K in the file. Consequently, the file grows or shrinks dynamically. We consider here the simplest form of queries that are based on exact key match. Thus, a hash file organization is the most appropriate in such an environment.

For a given key K a bucket that contains K is located using a family of hash functions h_i, $i = 0, 1, \ldots n$, that map a given key K to a bucket number, and an address table that maps a given bucket number to the number of the server that holds the bucket. The hash function h_i maps the key domain onto $B * 2^i$ addresses in the range $0, 1, \ldots, (B * 2^i - 1)$ where B is the initial number of buckets. Our file organization uses the following function:

$$h_i(K) = K \text{ mod } (B * 2^i)$$

With each bucket we associate a *bucket level* i, and we define the *server level* as the maximum bucket level located at the server. The bucket level is also an index for the hash function shown above. The highest server level among all servers is called the *file level* and denoted by L.

The *address table* contains the mapping of bucket numbers onto server numbers, and it contains the level of each bucket. An example of the address table is shown in Figure 1. This address table is similar to the directory of the extendible hashing method [FNPS79] except that it does not contain any "shared" entries for buckets at a level less than the file level. This compact format is advantageous when bucket levels can vary heavily for a given file (because of non-uniform insertions). To compute the

BucketNumber	0	1	2	3	4	5	6	7	12	15
BucketLevel	3	3	3	3	4	3	3	4	4	4
ServerNumber	5	2	2	4	1	4	5	3	3	1

Figure 1: **Example of the Address Table**

bucket to which a key is mapped, the file-level hash function h_L is applied to the key. If the computed bucket does not exist in the address table, this computation is repeated with decreasing level i of the hash function h_i until an existing bucket is returned. Finally, the table entry for this bucket yields the server number where the bucket resides. For example, the key 27 would be hashed to bucket 11 using h_4 if that bucket existed; but as bucket 11 does not exist in the address table, it is determined by using h_3 that the key belongs to bucket 3 which resides on server 4.

Each server has a specified performance capacity, and we assume, for simplicity, that this is the same for all servers. As we assume that the access load for a bucket is proportional to its size, the server performance capacity can be expressed in terms of the number of keys that a server can hold. Thus, we define the *feasible load capacity* C_F as the maximum number of keys that the server can keep without being overloaded. We assume that a server can be overloaded to some extent with degrading response time, before performance thrashing will eventually occur and response time will approach infinity because of queueing. Thus, we define the *panic load capacity* C_P as the number of keys after which a server is no longer capable of accepting additional keys. Buckets, on the other hand, are of variable size and can grow to any size provided the aggregate number of keys at a server does not exceed the panic load capacity of a server.

2.2 Client-Server Interaction

Each client c and each server s have their own perception of the file that is characterized by their value of the file level, denoted as L_c and L_s, respectively, and a copy of the address table that they currently have. The client's and the server's perception of the file may or may not coincide with the current picture of the file (i.e., they may have stale information on the file level and the address table).

When a client c invokes an operation on key K (e.g., to retrieve the record(s) with key K), it applies to K the hash functions h_i ($L_c \geq i \geq 0$) until it finds in its copy of the address table a server s where K should reside and sends K to that server.

When a server s receives a key K from the client, it applies to K the hash functions h_i ($L_s \geq i \geq 0$) until it finds in its copy of the address table a server s' where K

should reside as far as s is concerned. If K does belong to the server that received it (i.e., if $s = s'$), the server performs the requested operation. However, if K has been sent to the wrong server, the server forwards it to s', and updates the client's view with its file level L_s and its address table. Eventually, K is received by the correct server and the client operation is executed. Further, as an important side effect, clients gradually bring their perception of the file up-to-date, so that the probability of wrongly addressed messages is significantly reduced.

An interesting question is how many forwardings can happen. As we will see in our simulation results, the number of forwardings is indeed very small, and in the majority of cases no key required more than two forwardings. On the other hand, for any number t it is easy to design an experiment such that the length of a forwarding chain will exceed t. This can happen when the arrival rate of client requests is higher than the speed at which bucket splits can be performed. This fact, in a slightly different formulation, was pointed out in [Dev93]. In our experiments with the algorithm of [LNS93] we have observed cases where a key required more than two forwardings while the theoretical bound is only two forwardings. The discrepancy is caused by the fact that the derivation of the theoretical bound assumes a unrealistically synchronous sytem with instantaneous message delivery and instantaneous bucket splits.

2.3 Load Management

Our design is driven by two competing goals: to limit the growth in the number of servers across which the file is spread, and to guarantee good performance of each server by controlling the server load. The first goal is motivated by the cost minimization consideration; that is, we do not want to pay for a new server unless we have to for performance reasons. The second goal is motivated by performance considerations: we can efficiently service client requests only if none of the server utilizations is higher than the feasible load capacity. However, we assume that we have some slack in this respect by overloading a server temporarily (up to the server's panic load capacity), at the expense of degrading response time.

To reconcile these two goals, we allow redistributions of buckets, either by splitting or by migration of buckets. Such redistributions may be deferred if this is dictated by the cost objective up to the point when either all servers are operating in the range above the feasible load capacity or at least one server reaches the panic load capacity. In this case, and ideally only in this case, our method acquires a new server which would alleviate the existing servers by taking over one or more buckets. Following this rationale, the decision on increasing the system's resources is made dependent on the average server utilization. Based on our assumption that access load and data volume are proportional, we define the *global utilization* of the system as the ratio of

the average number of keys per server to the feasible load capacity of a server. Our consideration of cost/performance then amounts to the requirement that the global utilization should always be above some specified threshold U (e.g., $U \geq 0.9$) while also ensuring that no server is loaded higher than its panic load capacity would allow.

For monitoring and controlling the global utilization, we introduce a logically centralized agent that we call the *file advisor*. For simplicity, we will assume in this paper that the file advisor resides on a single dedicated server. Distributed implementations of the file advisor are conceivable, but are beyond the scope of this paper. As we will show in Section 5, however, the simplified implementation of the file advisor does not adversely affect scalability for fairly large systems.

One way of enforcing a global load control would be to require a server to report its load to the file advisor after each key that the server receives. Then, an additional server would be acquired only if the global utilization, with an additional server factored in, does not fall below U. However, this approach would significantly increase the message traffic between the advisor and the other servers, which we want to avoid. Therefore, we have adopted a different approach. Namely, we require a server to report its load to the file advisor by sending an *overload* message only if the server exceeds its feasible load capacity C_F. We piggyback on these messages information on the server's buckets, so that we can usually assume that the file advisor has knowledge of the up-to-date address table.

Once a server s has started to send *overload* messages, it continues to do so after each additional x keys that are added to the server (where x is a fine-tuning parameter with a typical value on the order of 10 or 100 depending on data and load characteristics) until the server receives either a *split* or *migrate* message from the file advisor.

Note that servers could incorporate the file advisor functionality themselves (possibly with less accurate estimations) rather than communicating to an explicit file advisor process; the file advisor is introduced for simplification of the algorithm.

Upon receiving an *overload* message from a server, the file advisor "*guesstimates*" the number of additional keys received by non-reporting servers since their last *overload* message and executes an adjustment procedure that updates the file advisor address table (see Section 3) We have developed a heuristic estimation method (described in Section 4) that is used by the adjustment procedure to come up with an "*educated guess*" of the current global utilization. A new server is acquired only if the current global utilization exceeds the specified threshold U.

When a new server is acquired, the file advisor selects the server s that has the highest number of keys among all existing servers and sends a split message to s

telling it that it should split all its buckets with a newly acquired server s'. A *split* message contains the address of the new server s' and the new address table for s. For each bucket b at s, records to be migrated to server s' are determined using h_{L_b+1} as the hashing function where L_b is the level of bucket b. After the split, the level L_b of bucket b at s and the new bucket at the new server s' are advanced to $L_b + 1$. The server levels of s and s' are updated accordingly. After s has completed a split it sends to the file advisor a *completion* message.

If a server s reaches the panic load capacity, then the file advisor first tries to migrate some of the buckets of s to other servers that have enough unused performance capacity to accept it. If that is possible and server s' is found that can accept at least one of the buckets b from s, the file advisor sends a *migrate* message to s. A *migrate* message contains bucket number b located at s, server address s' where b should be moved to, and the new address table. After s has completed bucket b migration to s' it sends to the file advisor a *completion* message.

If there is no server in the system that may accept any bucket from s, then a new server s' is acquired and s is instructed to split all its buckets with a newly acquired server using for each bucket b the hash function h_{L_b+1} where L_b is the level of bucket b at server s. After the split, the level L_b of each bucket b at s and the new bucket at the new server s' are advanced to $L_b + 1$. The levels of s and the new server s' are updated accordingly.

Since the file advisor does not have a precise information about the load at each server, its decision to migrate a bucket from server s to server s' in some cases cannot be implemented. It happens in the case, when the file advisor assumes that s' has lower load than s' really has. Recall, that s' has not reported an overload and therefore, the file advisor's information about the s' load is based on the real load estimate. Even if the estimate is quite accurate (as we will see in our performance results), from the time that s has sent a *panic* message to the time that s' receives a bucket from s, the server s' load could have increased and it will not be able to accomodate bucket b from server s.

If s' cannot accept b from server s, s reports to the file advisor that the migrate attempt was unsuccessful. There are many possible startegies to handle an unsuccessful migrate case. For simplicity, we selected here an option in which the file advisor receiving migration failed message, acquires a new server and sends a split message to s.

Pseudocode of the file advisor and the server algorithm is shown in figures 4 and 3, respectively.

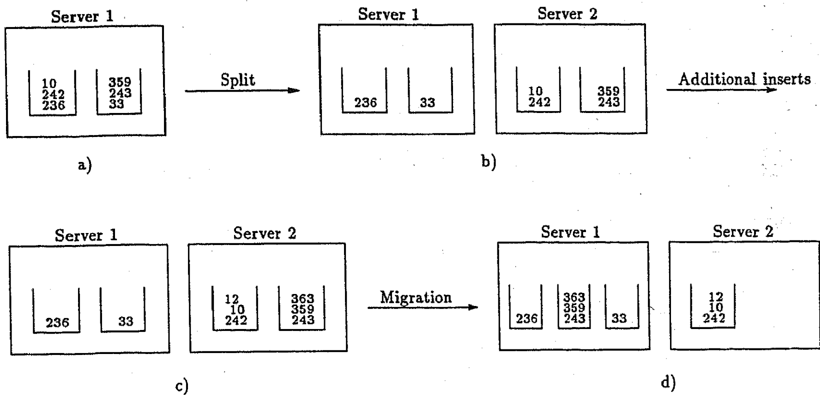

Figure 2: **Distributed File Configuration**
a. Server 1 sends OVERLOAD and receives SPLIT
b. File after SPLIT is performed
c. Additional keys 12 and 363 are inserted and server 2 is in PANIC
d. Bucket 2 from server 2 is migrated to server 1 to resolve PANIC

2.4 Example

Consider a distributed file where the servers feasible load capacity is 5 and its panic capacity is 6. Assume that the global utilization threshold U is 80% and initially the file has only one server with two buckets. Figure 2a depicts the file after it has received 6 keys. After receiving the 6th key, the server sends a panic message to the file advisor and since there are no more servers, a new server is acquired and all buckets at server 1 are split with server 2. Figure 2b shows the file after the split is performed.

After two additional keys are inserted at server 2, the server reports that it has reached the panic load capacity. The file advisor knows that the server received 2 extra keys and, after using a heuristic estimation (see Section 4), determines that no adjustment of the load at server 1 is required, and thus a migration of a bucket from server 2 to server 1 is performed. The file after the migration is shown in 2d.

```
Loop Forever;

  read next message;

   if (OPERATION REQUEST received from File Advisor) then
        if (correct address) then
             perform operation;
             send OPERATION ACKNOWLEDGEMENT to the client;
             if (overloaded) then
                  send OVERLOAD to the File Advisor;
        else
             forward message;
             if (first addressing error) then
                  send ADDRESS TABLE ADJUSTMENT to the client;

   else if (SPLIT received from File Advisor) then
        perform split of all buckets;
        send half of new buckets to the given new server;
        send SPLIT DONE to File Advisor;

   else if (MIGRATE received) then
       /* message contains BUCKET to be moved  to NEW_SERVER*/
        send BUCKET to NEW_SERVER;
        wait for response from NEW_SERVER;
        if (BUCKET ACCEPTED received from NEW_SERVER) then
             delete BUCKET;
             send MIGRATE DONE to File Advisor;
        else
             send MIGRATE REJECTED to File Advisor;
             wait for SPLIT from File Advisor;

   else  if (NEW BUCKET received from another SERVER) then
        if (enough free capacity to accept the bucket) then
             insert the bucket;
             send BUCKET ACCEPTED to SERVER;
        else send BUCKET REJECTED to SERVER;
End;
```

Figure 3: Server's algorithm

```
Loop Forever;

    read next message;

    if (OVERLOAD received from SERVER) then
         update address table with the received info;
         update address table with the estimates;

         if (global utilization after split >= threshold $U$) then
              send SPLIT to MOST_OVERLOADED_SERVER;

         else if (Panic_Mode flag is on) then
              Send MIGRATE to SERVER;
              /* move bucket that fits into 1/2 of the free capacity
                 of the least utilized server */

    else if (SPLIT DONE or MIGRATE DONE received) then
         update address table;

    else if (MIGRATE REJECTED received from SERVER) then
         Send  SPLIT to SERVER;
End;
```

Figure 4: File Advisor's algorithm

3 Tracking the System Load

The file advisor maintains information about the number of keys at each server. Because an overloaded server keeps the advisor up-to-date about its load, the advisor has precise information about the current number of keys at an overloaded server. For non-overloaded servers, however, the file advisor maintains an estimate of the number of keys at each server. As time passes and inserts take place, some server may become overloaded whereas other servers may still absorb inserts without getting overloaded. To facilitate a global load control, the file advisor must keep a good estimate of the total number of keys at non-overloaded servers.

Whenever the file advisor receives an *overload* message, it adjusts its estimate of the number of keys at non-overloaded servers to take into account possible inserts in their buckets. Assume that server s reports that it has become overloaded and reports its total number of keys and their distribution among its buckets. The file advisor now

derives that s has received t extra keys since it last adjusted the number of keys of server s.

The file advisor, based on its address table information, computes the total number of buckets N as well as the number of non-overloaded buckets R as if all buckets were at the file level L. This simplifying assumption is feasible as a bucket at level i can be viewed as being equivalent to two buckets at level $i+1$. For example, if the file contains 3 buckets of level 3 and the file level is 5, then the file advisor considers that there are 12 buckets of level 5. It then uses the heuristic procedure described in Section 4 to obtain the expected number of keys j obtained by non-overloaded buckets (and by implication, by non-overloaded servers). If j is less than one, then we assume that none of the non-overloaded buckets has received any additional keys.

In the latter case, we do not completely discard the information that s has reported to the file advisor. Clearly, with the next overload message reporting that server s' (which may or may not be the same as s) has received t_1 additional keys, the probability that non-overloaded buckets have received some extra keys is increased, provided that keys are uniformly distributed. Therefore, when the server s' reports that it has received t_1 extra keys, the file advisor uses an estimation heuristics not for t_1 but for $t + t_1$, to account for the information that has not yet been used for adjusting the estimated number of keys.

The file advisor then updates its estimate of a number of keys at non-overloaded servers by "distributing" these j keys among them. For the purpose of the distribution, the non-overloaded servers are processed in an non-decreasing order of the number of keys at them as follows:

1. Let server s have r buckets at the file level L. (For example, if server has 3 buckets at level 3 and one bucket at level 4 and the file level is 5, then the file advisor assumes that the server has a total of $3 * 2^{5-3} + 1 * 2^{5-4} = 14$ buckets at level 5). Then the file advisor increases its current estimate of s by r.
2. Decrease j by r and continue with the next non-overloaded server if j is still greater than 0.

To illustrate the above adjustment procedure, let us consider the following example. Assume that the file consists of 4 servers and has the file level $L = 4$. Two servers contain one bucket each and the other two servers contain two buckets each. Let the feasible load capacity of a server be 20 keys. Assume that the key distribution among buckets and the buckets levels are as follows (such a configuration could have been produced, for example, by first inserting many keys and then deleting a large fraction of them):

- Server 1: containing 23 keys in 1 bucket of level 3;
- Server 2: containing 6 keys in 1 bucket of level 4;
- Server 3: containing 4 keys in two buckets: one bucket of level 3 containing 2 keys and one bucket of level 4 containing 2 key;
- Server 4: containing 4 keys in two buckets: one bucket of level 4 containing 3 keys and one bucket also of level 4 containing 1 key.

Thus, server 1 is overloaded while servers 2, 3, and 4 are non-overloaded. The file advisor knows the precise number of keys at server 4, while for servers 2, 3, and 4 the shown numbers are the best estimates that the advisor has come up with so far.

Assume now that the file advisor has received an overload message from server 4 indicating that the server has received 17 more keys with bucket 1 receiving 8 keys and bucket 2 receiving 9 keys. Thus, server 4 has become overloaded. The file advisor calculates that there are altogether 8 buckets of level 4 and among them 6 buckets of level 4 that are non-overloaded (server 2 contains one such bucket, server 3 contains 1 bucket of level 3 which is perceived by the file advisor as 2 buckets of level 4, and one more bucket of level 4, etc.)

Using the estimation heuristics of Section 4, the adjustment procedure derives that the number of keys in servers 2 and 3 needs to be adjusted by a total of 4 keys. Observe that at this point the file advisor knows the precise number of keys at servers 1 and 4. The file advisor adjusts the number of keys at server 2 by increasing it by 1 and the number of keys at server 3 again by increasing it by 3.

4 Estimation Heuristics

In this section we discuss the heuristic method used by the file advisor to estimate the number of keys received by non-overloaded servers at the time that some server has reported an overload. Let the additional number of keys reported by the overloaded server be denoted by t. Based on the value of t, the file advisor should estimate the number of keys that were received by the non-overloaded servers. Recall that we assume that the access load of a server is proportional to the number of keys that it holds. Thus, the estimate of the received extra keys reflects the increase of the access load.

The heuristics we propose here assumes an uniform distribution of keys across buckets (provided that buckets were at the same level). Assume that the file has level L and

contains n buckets in total. Since a bucket with level i contains, on average, twice as many keys as a bucket with level $i+1$, we can simplify the estimation by viewing the level i bucket as two the level $i+1$ buckets. In the sequel, we will therefore assume, for the sake of a simpler presentation, that there are $N = 2^L$ buckets in the system all of which are at the same level L (where L is the file level). Thus, the probability for a key to be placed into a given bucket is $1/N$.

Recall that once a server has sent an overload message to the file advisor, it continues to do so after every additional x keys. Thus all servers in the file are subdivided into *overloaded* and *non − overloaded* servers. All buckets at the overloaded and non-overloaded servers are referred to as overloaded and non-overloaded buckets, respectively. Thus, all N buckets of the same level L are also subdivided into overloaded and non-overloaded buckets.

Now, the problem we need to address can be formulated as follows: Assume that there are N buckets with R of them non-overloaded and S of them overloaded ($N = R+S$). Furthermore, assume that we know that one of these buckets has reported that it received t keys. We also know that none of the overloaded buckets (except may be the one that has reported!) has received any keys. We need to determine the number of keys j received by the R non-overloaded buckets.

Let s be the total number of keys in overloaded buckets and r be the total number of keys in non-overloaded buckets at the time of the last adjustment for the non-overloaded servers. Since none of the non-overloaded buckets reports its load to the file advisor, the total number of keys at all such buckets cannot exceed $R * C_F$, where C_F is the feasible load capacity of a server. On the other hand, we know that after a server has reported that it has received t additional keys, the number of keys in overloaded buckets is precisely $s + t$.

Now, instead of having $N = S + R$ overloaded and non-overloaded buckets, we assume that we have only 2 "metabuckets". One of them contains keys from all non-overloaded buckets (we denote it by A) and the other contains keys from all overloaded buckets (we denote it by B). From the information available to the file advisor we can derive the probability p that metabucket A receives a key and the probability q that metabucket B receives a key:

$$p = \frac{R}{N} \qquad q = \frac{S}{N}$$

In the sequel $< a, b >$ stands for a configuration, where a is the number of keys in metabucket A and b is the number of keys in metabucket B. At the time that a bucket has reported an overload, we know that the last key has been received by an

overloaded bucket. Thus, if there are no additional constraints on configurations, the probability of each configuration can be easily determined as follows:

$$Prob(< a, b >) = \binom{a+b-1}{a} p^a q^{b-1}$$

However, the configurations are subject to the following constraints:

1. $r \leq a \leq R * C_F$.

2. $b = s + t$.

Thus, we have to evaluate a conditional probability of a given configuration subject to the above conditions. The probability that the above conditions are satisfied is as follows:

$$M = \sum_{j=r}^{R*C_F} Prob(< j,\ s+t >) = \sum_{j=r}^{R*C_F} \binom{s+t+j-1}{j} p^j q^{s+t-1}$$

Thus, our elementary event space consists of $R * C_F - r + 1$ different events where each event has a probability of $Prob(< a,\ b >)\ /\ M$.

Let $EV(t,\ R)$ be the expected value of the number of keys received by the R non-overloaded buckets at the time that some server has reported that it has received t additional keys. Thus, the following formula holds for $EV(t,\ R)$:

$$EV(t, R) = \sum_{j=r}^{R*C_F} j \frac{Prob(<j,\ s+t>)}{M} = \sum_{j=r}^{R*C_F} j \frac{\binom{s+t+j-1}{j} p^j q^{s+t-1}}{M}$$

$EV(t,\ R)$ can be now calculated precisely using the formulas given above. However, the calculation of $EV(t,\ R)$ should be done very efficiently. The formula indicates, however, that the precise calculation of $EV(t,\ R)$ is computationally expensive. Thus, we actually use the approximation

$$EV(t,\ R) \approx \frac{(t-1)(R-1)}{2N}$$

for the value of $EV(t,\ R)$. This approximation is both computationally inexpensive and fairly accurate (as determined by comparing the exact and approximate figures for a large range of p and q values). We use this formula as an approximation for all possible values of p and q.

5 Preliminary Simulation Results

We have been conducting a performance evaluation of our algorithm using a discrete-event simulation model. This section reports some preliminary results of this study. The simulation model was built using the CSIM run-time library [Sch92]. Our model simulates a distributed system consisting of a group of clients and servers connected by a network.

5.1 Simulation Model

Our simulation model consists of three major components: a server model, a client model and a network model. Each server has a CPU and disk storage with identical characteristics. In general each server may have a different local data organization. For example, one server may use a B^+-tree organization while another server may use dynamic hashing. Thus, the CPU and I/O costs of access data may vary from server to server. The simulation model is able to account for that; however, in the experiments described below we modelled a homogeneous system where all servers use a hash-based organization locally.

The feasible load capacity of a server is expressed in terms of the maximum number of keys that the server is able to keep (and service their access requests) without any performance degradation. Similarly, the panic load capacity of a server is expressed in terms of the maximum number of keys after which the server would start thrashing. Each server has the same I/O block size (set to the size of a disk track); the record size and the key size were fixed throughout the simulation. For simplicity, we assume that queries do not benefit from caching: each query is assumed to cause exactly one disk I/O. Inserts, on the other hand, are assumed to be batched, so that a number of inserted records can be written to disk in a single disk I/O. Logging is employed to ensure that inserts are not lost (due to server failures) before they are eventually written to disk. Bucket splits and migrations are implemented by large multi-block disk I/Os for disk efficiency. Both the disk I/Os and the network traffic of splits and migrations are implemented as background processes.

Each client randomly generates keys and insert operations or queries on them to be sent to the servers. Keys are generated using a uniform distribution. The arrivals of client requests are exponentially distributed with the same average arrival rate for each client. Each client request is acknowledged. However, the acknowledgement is asynchronous with requests submissions.

server instruction rate	10 MIPS
single-block disk I/O time	20 msec
record size	10 KBytes
key size	100 Bytes
block size	50 KBytes
feasible load capacity	10000 records
panic load capacity	11000 records
CPU cost for servicing a client request	10000 instructions
client requests arrival rate	0.1 requests/sec per client
fraction of insert requests	0.1
fraction of queries	0.9
network latency	20 μsec
network bandwidth	10 MBytes/sec
CPU cost for each message	5000 instructions
maximum packet size	1 MByte

Figure 5: **Setting of Simulation Parameters**

The network is modelled assuming that at each time there are no more than t packets being transmitted through the network. The experiments were conducted for $t = 1$. The network performance in the model was characterized by the network latency NL (i.e., the time required to process each packet, independently of the packet size) and the bandwidth BW (i.e., the transmission time that is needed to send one byte through the network). The total time each packet spends in the network is equal to: $NL + BW * PacketSize$. The large data transfers that are caused by redistribution of buckets are divided up into a number of packets with a specified maximum size. All other messages correspond to exactly one packet.

5.2 Simulation Experiments and Results

We have conducted a number of experiments for different system and workload parameters. Since they all showed consistent results without significant differences, we concentrate here on a single series of experiments in which all system and workload parameters were kept invariant. The values of these parameters are summarized in Figure 5. These settings were chosen to model approximately a workstation farm with an FDDI interconnect.

The "cost-conscious" scalability of the algorithm is demonstrated as follows. We first loaded a distributed file of a specific size by issuing only insert requests from a specific number of clients, such that the final file size would be proportional to the

# clients	final # servers	# servers splits	# bucket migrations	% requests w/o forward	max. forward
100	11	10	21	99.97 %	1
200	21	20	44	99.97 %	1
300	33	32	73	99.97 %	1
500	53	52	129	99.97 %	1
1000	107	106	243	99.97 %	1

Figure 6: **Results for the Loading Phase of the Experiments (for $U = 0.9$)**

number of clients. The loading was done by employing our algorithm, starting from a single ("empty") server having ten buckets and acquiring servers as dictated by our redistribution method. The number of splits and migrations during this loading phase and the resulting network traffic is shown in Figure 6.

Note that the correlation between the eventual file size and the number of clients (and thus also the access rate) follows the type of scaling rules that are used in various transaction processing and database benchmarks [Gr91]. Specifically, each client would insert 1000 records (with a total data volume of 10 MBytes), so that, for example, 100 clients correspond to a file size of 100000 records (1 GByte). Note that the data size itself is used here only to represent a proportional access load, and that such relatively small figures can themselves be scaled up by keeping the ratio of clients and file size constant.

After the loading phase, we ran a mix of insert requests and queries (with a ratio of 1 to 9, see Figure 5) from the same number of clients until the file size had grown by 10 percent of the file size as it was right after the loading. For example, with 100 clients and a file size of 100000 records, a simulation run included 10000 insert requests (plus 90000 queries). The simulation results given below were collected during this execution phase.

To demonstrate scalability, we repeated the described experiment (both the loading and the execution phase) for different numbers of clients and corresponding file sizes, ranging from 100 clients (100000 records, 1 GByte) to 1000 clients (1 Mio. records, 10 GBytes). The global utilization threshold U was set to 0.9 in all runs; that is, the goal was to limit the number of servers such that the average server load would be at least 90 percent of the feasible load capacity while also ensuring that no server would have a load higher than the panic load capacity. The main results for this series of experiments are given in Figures 7 and 8, separated into performance-oriented and cost-oriented metrics.

The most striking result of the experiments was that our algorithm did indeed manage

# clients	total throughput	avg. resp. time of inserts	avg. resp. time of queries	% requests w/o forward	max. forward
100	10 requests/sec	1.80 msec	22.8 msec	99.5 %	2
200	20 requests/sec	1.88 msec	22.9 msec	99.2 %	2
300	30 requests/sec	1.89 msec	22.9 msec	99.3 %	2
500	50 requests/sec	2.13 msec	23.2 msec	98.8 %	2
1000	100 requests/sec	2.49 msec	23.7 msec	98.7 %	3

Figure 7: **Performance Results of the Simulation Experiments (for $U = 0.9$)**

# clients	# servers	# buckets	avg. server utilization	# bucket splits	# bucket migrations	failed migrations
100	12	134	0.91	1	1	1
200	24	251	0.91	3	3	0
300	36	396	0.91	3	4	3
500	59	617	0.93	6	19	6
1000	119	1340	0.92	12	20	2

Figure 8: **Cost Results of the Simulation Experiments (for $U = 0.9$)**

to keep the global utilization above 90 percent, while also providing almost constant response time for queries (which are more critical than inserts) for linearly increasing throughput. Thus, we have a constant cost/performance ratio, as expressed, for example, in the ratio of the number of clients (which is proportional to throughput) to the number of servers upon which the file is spread. Further note that the number of servers and buckets also grows only linearly with the file size and access load; this shows that the splitting of servers is indeed done in a carefully controlled way. Furthermore, in most cases splits were caused by failed attempts to migrate buckets (which failed because no other could accept the additional load without becoming overloaded itself). This is a reconfirmation that our algorithm maintains a good cost/performance ratio, so that each acquired server is utilized at an acceptable level.

The good response time result can be attributed to two observed effects. First, as shown in Figure 7, almost all client requests could be serviced without any forwarding, and the longest chain of forwardings was 2 for most cases and 3 (for a few requests) in the case of 1000 clients. Secondly, as shown in Figure 8, the number of bucket splits and migrations that would potentially cause some delays in the servicing of client requests (because of disk or network contention) was small enough so as not to have any significant adverse effect on the response time of client requests. The mild increase in the response time with increasing file size is indeed caused by the interference of client requests and bucket redistribution, however. On the other hand,

# clients	total # messages	# overload messages	avg. # packets per split / migration
100	203,967	3084	40.5
200	415,425	12658	40.5
300	613,689	9727	35.3
500	1,028,184	17248	25.2
1000	2,063,260	36978	33.1

Figure 9: **Network Costs of the Simulation Experiments** (for $U = 0.9$)

recall that the file size was increased by 10 percent during the measurement phase, so that some interference is inescapable. Finally, note that insert requests have a much shorter response time than queries because they are written to disk in batches and an acknowledgement is sent to the client already after having logged the insert, whereas queries need to fetch a record from disk.

The network had a low utilization in all experiments and never incurred any bottleneck. A summary of the message costs in the execution phase is given in Figure 9 for completeness. Note that the vast majority of messages simply correspond to the client requests and server responses (i.e., they do not represent any additional overhead). The network traffic due to splits and migrations (which are the only larger messages) was fairly low.

The described type of scalability experiment was performed also for other values of the global utilization threshold U (0.8 and 0.7); these experiments merely confirmed the above findings and are thus omitted here. To conclude this section, we observe that the variance of figures provided in the above tables is quite low, which gives us a high confidence level of our experimental figures.

6 Conclusion

In this paper we have presented a new distributed file organization that supports dynamic growth in terms of both file size and access load while allowing us to control the cost/performance ratio of the distributed system. Unlike previous approaches to scalable distributed file organizations that do not have the kind of "cost-consciousness" that we are advocating, our approach acquires a new server only if the global utilization of servers does not drop below a specified threshold while also ensuring that no server is overloaded. Thus, we minimize the number of servers that are needed to sustain the required performance. This is an important achievement as the system administration and the additional steps for ensuring high availability (that would,

perhaps, be necessary but are disregarded in this paper) incur significant costs in proportion to the number of servers that are involved.

The presented simulation experiments show very promising scalability results, but are still too preliminary to draw any final conclusions. We are in the process of performing a comprehensive simulation study. We also plan to compare our approach to other recently proposed methods for distributed hash files, notably the methods of [LNS93] and [Dev93]. Note, however, that these methods in their original form are not really comparable to our approach, as they disregard the important issue of cost/performance. This issue has to be added to these previous approaches (i.e., some form of controlling the global utilization) in order to conduct a systematic comparison.

Beyond our current system model with homogeneous servers, our approach has the potential of being applicable also to heterogeneous servers where servers may differ in their local data organization or may have different load capacities. Another extension of our approach would be to replace the logically centralized file advisor process by a distributed algorithm that would be carried out by the servers themselves. These extensions are certainly feasible, and details are being worked out. Finally, we are working also on adding controlled redundancy to the file organization to enhance data availability in the presence of server failures.

References

[BS85] A. Barak, A. Shiloh, A Distributed Load Balancing Policy for a Multicomputer, Software Practice & Experience Vol.15 No.9, September 1985, pp. 901-913.

[CS92] D.D. Chamberlin, F.B. Schmuck, Dynamic Data Distribution (D^3) in a Shared-Nothing Multiprocessor Data Store, VLDB Conference, Vancouver, 1992.

[Dev93] R. Devine, Design and Implementation of DDH: A Distributed Dynamic Hashing Algorithm, 4th International Conference on Foundations of Data Organization and Algorithms (FODO), Chicago, 1993.

[DG92] D.J. DeWitt, J.N. Gray, Parallel Database Systems: The Future of High Performance Database Systems, Communications of the ACM Vol.35 No.6, June 1992, pp. 85-98.

[ED88] R.J. Enbody, H.C. Du, Dynamic Hashing Schemes, ACM Computing Surveys Vol.20 No.2, June 1988, pp.85-113.

[ELZ86] D.L. Eager, E.D. Lazowska, J. Zahorjan, Adaptive Load Sharing in Homogeneous Distributed Systems, IEEE Transactions on Software Engineering Vol.12 No.5, May 1986, pp. 662-675.

[FNPS79] R. Fagin, J. Nievergelt, N. Pippenger, H.R. Strong, Extendible Hashing - A Fast Access Method for Dynamic Files, ACM Transactions on Database Systems Vol.4 No.3, 1979, pp. 315-344.

[Gr91] J. Gray (Editor), The Benchmark Handbook for Database and Transaction Processing Systems, Morgan Kaufmann, 1991.

[HW94] Y. Huang, O. Wolfson, Object Allocation in Distributed Databases and Mobile Computers, Data Engineering Conference, Houston, 1994.

[JK93] T. Johnson, P. Krishna, Lazy Updates for Distributed Search Structure, ACM SIGMOD Conference, Washington, 1993.

[Lar88] P.A. Larson, Dynamic Hash Tables, Communications of the ACM Vol.31 No.4, April 1988, pp. 446-457.

[LLM88] M.J. Litzkow, M. Livny, M.W. Mutka, Condor - A Hunter of Idle Workstations, 8th International Conference on Distributed Computing Systems (DCS), San Jose, 1988.

[LNS93] W. Litwin, M.-A. Neimat, D.A. Schneider, LH* - Linear Hashing for Distributed Files, ACM SIGMOD Conference, Washington, 1993; extended version published as: Technical Report HPL-93-21, Hewlett-Packard Labs, 1993.

[MS91] G. Matsliach, O. Shmueli, An Efficient Method for Distributing Search Structures, 1st International Conference on Parallel and Distributed Information Systems (PDIS), Miami Beach, 1991.

[Sch92] H. Schwetman, CSIM Reference Manual (Revision 16), Microelectronics and Computer Technology Corporation, Austin, 1992.

[WDJ91] C.B. Walton, A.G. Dale, R.M. Jenevein, A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins, VLDB Conference, Barcelona, 1991.

[WJ92] O. Wolfson, S. Jajodia, Distributed Algorithms for Dynamic Replication of Data, ACM PODS Conference, San Diego, 1992.

[WSZ91] G. Weikum, P. Scheuermann, P. Zabback, Dynamic File Allocation in Disk Arrays, ACM SIGMOD Conference, Denver, 1991.

Eidg. Techn. Hochschule Zürich
Informatikbibliothek
ETH-Zentrum
CH-8092 Zürich