Distributed File Organization with Scalable Cost/Performance*

Radek Vingralek† Yuri Breitbart† Gerhard Weikum‡

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 significant and challenging problem addressed here is how to achieve scalability so that both the file size and the client throughput 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 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., [SPW90, 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 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. 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

*This material is based in part upon work supported by the grant from Hewlett-Packard Corporation and by NSF under grant IRI-92121301.

†Department of Computer Science, ETH Zurich, CH-8092, Switzerland (on sabbatical leave from Department of Computer Science, University of Kentucky, Lexington, KY 40506).

‡Department of Computer Science, ETH Zurich, CH-8092, Switzerland.

SIGMOD 94- 5/94 Minneapolis, Minnesota, USA
© 1994 ACM 0-89791-639-5/94/0005..$3.50

utilization includes both storage and performance capacity. 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 number of buckets grows. Thus, other applications cannot really count on this "free" capacity.
- More servers imply a higher cost in terms of system administration and availability, 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.
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 in redistributing the load of an overloaded server. We can redistribute the load either by *splitting* one or more buckets of the server or by *migrating* some of its buckets to another server. 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 represents 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 occupancy. 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 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). The only exception from this policy is in the case that there is no server that a bucket can migrate to and yet one of the servers reached a maximum capacity that started to affect its performance.

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., [RS84, ED88, Lar88]). We assume, for simplicity, that the access load on 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 uti-

lization. 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. (see also [RS84]). 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, Section 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. We consider here the simplest form of queries that are based on exact key match.

For a given key K a bucket that contains K is located using a family of hash functions h_i, $i = 0, 1, \ldots$, 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 $B \cdot 2^i$ addresses in the range $0, 1, \ldots, (B \cdot 2^i - 1)$ where B is the initial number of buckets. Our file organization uses the following function:

$$h_i(K) = K \bmod (B \cdot 2^i)$$

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.

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

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

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 format is advantageous when bucket levels can vary heavily for a given file (because of non-uniform insertions). To compute the 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 an unrealistically synchronous system 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 that alleviates the load on 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.

The file advisor maintains information about the number of keys at each server in its address table. 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 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.

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. If the current global utilization exceeds the specified threshold U, a new server is acquired

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'. 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.

If a server s reaches the panic load capacity, but the global utilization would not exceed the threshold after the split, 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. 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 s'.

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 its previosly reported load. 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 accommodate 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 migration. 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 for actions executed by the client, the server and the file advisor is shown in figures 2, 3 and 4, respectively.

```
Loop Forever;

   if (key to be sent) then
      determine address;
      send key to the address;

   if (ADDR_TABLE_ADJUST received
     from server) then
      /* message contains new address table */
      set address table to new table;

   if (OPER_ACK received from server) then
      process operation acknowledgement;

End;
```

Figure 2: **Client's algorithm**

```
Loop Forever;

   read next message;

   if (OPER_REQ received from client) then
      if (correct address) then
         perform operation;
         send OPER_ACK to client;
         if (overloaded) then
            send OVERLOAD to File Advisor;
      else
         forward message;
         if (first addressing error) then
            send ADDR_TABLE_ADJUST to client;

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

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

   else if (NEW BUCKET received
          from SERVER) then
      if (enough storage to accept bucket) then
         insert the bucket;
         send BUCKET_ACCEPT to SERVER;
      else send BUCKET_REJECT to SERVER;
End;
```

Figure 3: **Server's algorithm**

2.4 Example

Consider a distributed file where each server's 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 5a 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 5b 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. Figure 5c shows the file after insertion of the two keys. The file advisor knows that the server received 2 extra keys and, after using a heuristic estimation (described later in 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 figure 5d.

```
Loop Forever;

   read next message;

   if (OVERLOAD received from SERVER) then
      update Addr Table  with received info;
      update Addr Table  with estimates;

      if (load factor after split >=
         load factor threshold) then
         send SPLIT to MOST_OVERLOAD_SERVER;

      else if (Panic_Mode flag is on) then
         Send MIGRATE to MOST_EMPTY_SERVER;
         /* move bucket that fits into      */
         /* 1/2 of space on emptiest server */
            .
   else if (SPLIT_DONE or MIGRATE_DONE
          received from SERVER) then
      update Addr Table;

   else if (MIGRATE REJECT 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's information about the current number of keys at an overloaded server may differ from the actual load of such server by no more than $x - 1$, where x is a number of keys that after receiving which an overloaded server reports its load again to the file advisor. 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.

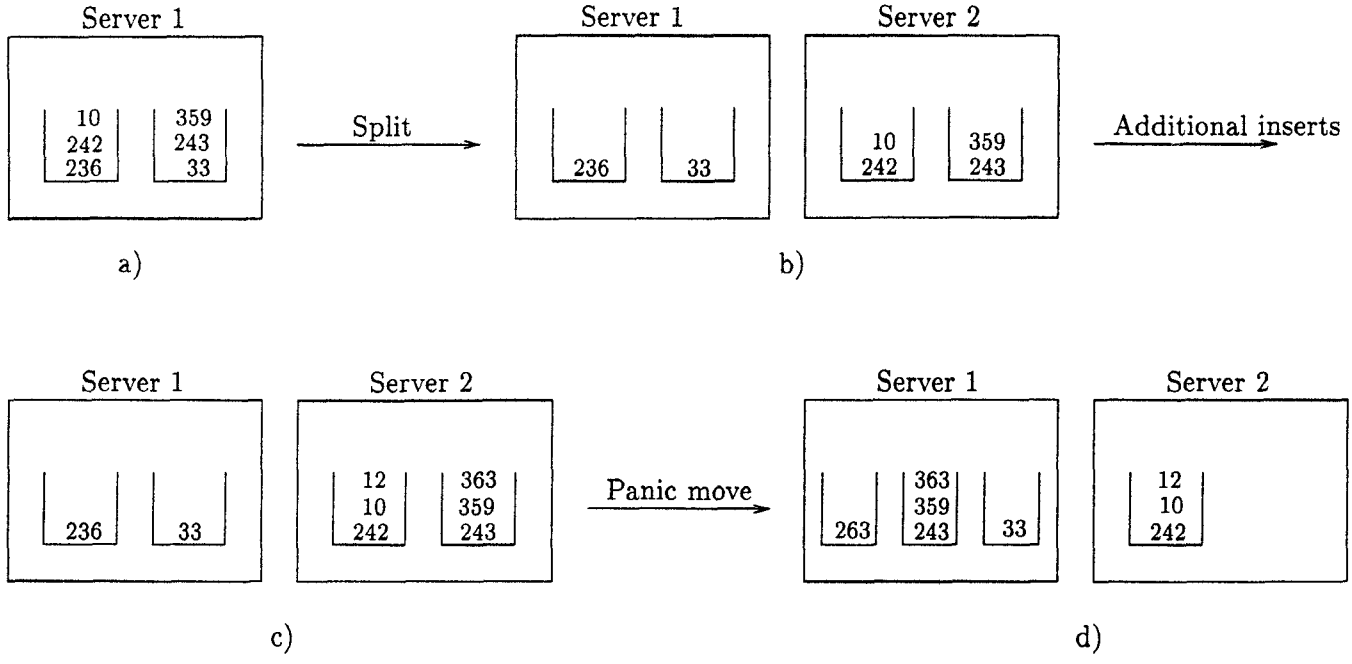

Figure 5: **Distributed File Configuration**
a) Server 1 sends OVERLOAD and receives SPLIT
b) File after SPLIT is performed
c) Additional keys 14 and 363 are inserted and server 2 is in PANIC
d) Bucket 2 from server 2 is migrated to server 1 to resolve PANIC

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 $E(t)$ obtained by non-overloaded buckets (and by implication, by non-overloaded servers). If $E(t)$ 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 $E(t)$ keys among them. For the purpose of the distribution, the non-overloaded servers are processed in a 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 $3 \cdot 2^{5-3} + 1 \cdot 2^{5-4} = 14$ buckets at level 5). Then the file advisor increases its current estimate of s by j (without exceeding feasible capacity of the server), where $j = min(r,\ E(t))$.

2. Decrease $E(t)$ by r and continue with the next non-overloaded server if $E(t)$ 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 file level $L = 4$. The feasible load capacity of each server is 20 keys. Assume the key distribution is as given in Figure 6 (such a configuration could have been produced, for example, by first inserting many keys and then deleting a large fraction of them).

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 1, while for servers 2, 3, and 4 the shown numbers are the best estimates by the advisor

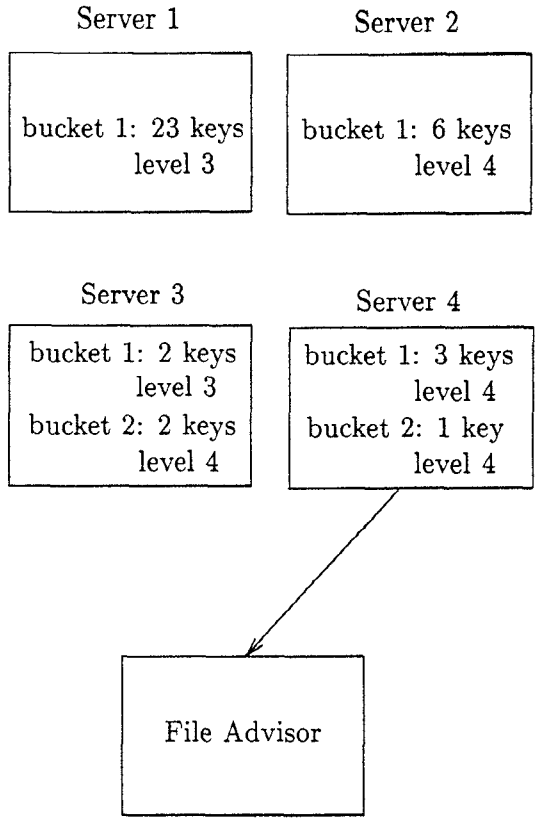

Figure 6: **File Load Adjustment Example**

Assume now that the file advisor has received an overload message from server 4 indicating that the server has received $t = 17$ more keys. Thus, server 4 has become overloaded. The file advisor calculates that there are altogether $N = 8$ buckets of level 4 and among them $R = 4$ 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 also contains one bucket of level 4).

Using the estimation heuristic of the next Section, the adjustment procedure derives that the number of keys in servers 2 and 3 needs to be adjusted by a total of $E(17) = 4$ 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.

The heuristic 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 buckets of level $i+1$. In the sequel, we will therefore assume, 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. To simplify our discussion, we assume that $x = 1$. 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. Let R be the number of non-overloaded buckets among all N 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 need to determine the number of keys $E(t)$ received by non-overloaded buckets (recall, that under our assumption, no overloaded bucket (except the one that is reporting an overload) has received any extra keys!).

Let 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, the total number of new keys received at all such buckets cannot exceed $R \cdot C_F - r$, where C_F is the feasible load capacity of a server.

Now, 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 derive the probability p that metabucket A receives a key and the probability q that metabucket B receives a key as $p = R/N$ and $q = S/N$.

In the sequel $< j, t >$ stands for a configuration, where j is the number of new keys received in metabucket A at the time that metabucket B reported that it has received t additional keys. 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 is as follows:

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

However, the configurations are subject to the following constraint:

$$0 \leq j \leq R \cdot C_F - r$$

The probability that the above condition is satisfied is:

$$\begin{aligned} M &= \sum_{j=0}^{R \cdot C_F - r} Prob(< j,\ t >) \\ &= \sum_{j=0}^{R \cdot C_F - r} \binom{t+j-1}{j} p^j q^t \end{aligned}$$

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

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

$$\begin{aligned} E(t) &= \sum_{j=0}^{R \cdot C_F - r} j \frac{Prob(< j,\ t >)}{M} \\ &= \sum_{j=0}^{R \cdot C_F - r} j \frac{\binom{t+j-1}{j} p^j q^t}{M} \end{aligned}$$

The formula indicates, however, that the precise calculation of $E(t)$ is computationally expensive. Thus, we actually use the approximation

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

for the value of $E(t)$. This approximation is fairly accurate (as determined by comparing the exact and approximate figures for a large range of p and q values).

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].

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, and uses a hash-based file 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 (this can be done irrespective of the insert rate provided the inserts are acknowledged immediately and performed on the background). Logging is employed to ensure that inserts are not lost (due to server failures) before they are eventually written to disk.

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.

The network performance in the model was characterized by the network latency NL and the bandwidth BW The total time each packet spends in the network is equal to: $NL + BW \cdot PacketSize$. 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.

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 7: **Setting of Simulation Parameters**

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 7. 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 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 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 7) 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 Million 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 8 and 9, separated into performance-oriented and cost-oriented metrics.

# 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 8: **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 9: **Cost Results of the Simulation Experiments (for $U = 0.9$)**

The most striking result of the experiments was that our algorithm did indeed manage 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 server 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 8, 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 9, 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, recall that the file size was increased by 10 percent during the measurement phase, so that some interference is inescapable.

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 10 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 measurments provided in the above tables is quite low, which gives us a high confidence level of our experimental figures.

# clients	total # messages	# overload messages	avg. # packets per split or 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 10: **Network Costs of the Simulation Experiments (for $U = 0.9$)**

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.

[RS84] K. Ramamohanarao, R. Sacks-Davis, Recursive Linear Hashing, ACM Transactions on Database Systems, Vol.9 No.3, September 1984, pp. 369-391.

[SPW90] C. Severance, S. Pramanik, P. Wolberg, Distributed Linear Hashing and Parallel Projection in Main Memory Databases, VLDB Conference, Brisbane, 1990.

[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.