LH* — A Scalable, Distributed Data Structure

Witold Litwin
Université Paris 9 Dauphine
and
Marie-Anne Neimat
Hewlett-Packard Laboratories
and
Donovan A. Schneider
Red Brick Systems

We present a scalable distributed data structure called LH*. LH* generalizes Linear Hashing (LH) to distributed RAM and disk files. An LH* file can be created from records with primary keys, or objects with OIDs, provided by any number of distributed and autonomous clients. It does not require a central directory, and grows gracefully, through splits of one bucket at a time, to virtually any number of servers. The number of messages per random insertion is one in general, and three in the worst case, regardless of the file size. The number of messages per key search is two in general, and four in the worst case. The file supports parallel operations, e.g., hash joins and scans. Performing a parallel operation on a file of M buckets costs at most $2M + 1$ messages, and between 1 and $O(\log_2 M)$ rounds of messages.

We first describe the basic LH* scheme where a coordinator site manages bucket splits, and splits a bucket every time a collision occurs. We show that the average load factor of an LH* file is 65–70% regardless of file size, and bucket capacity. We then enhance the scheme with load control, performed at no additional message cost. The average load factor then increases to 80–95%. These values are about that of LH, but the load factor for LH* varies more.

We next define LH* schemes without a coordinator. We show that insert and search costs are the same as for the basic scheme. The splitting cost decreases on the average, but becomes more variable, as cascading splits are needed to prevent file overload. Next, we briefly describe two variants of splitting policy, using parallel splits and presplitting that should enhance performance for high-performance applications.

All together, we show that LH* files can efficiently scale to files that are orders of magnitude larger in size that single-site files. LH* files that reside in main memory may also be much faster than single-site disk files. Finally, LH* files can be more efficient than any distributed file with a centralized directory, or a static parallel or distributed hash file.

Categories and Subject Descriptors: D.4.3 [**Software Engineering**]: File Systems Management

---

Witold Litwin's work was done while the author was visiting HP Laboratories. Donovan Schneider's work was done while the author was at HP Laboratories.
Authors' addresses: Witold Litwin: Université Paris 9 Dauphine, Place du Marechal de Lattre De Tassigny, 75775 Paris Cedex 16, France, email: <litwin@etud.dauphine.fr>; Marie-Anne Neimat: Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, California, 94304, USA, email: <neimat@hpl.hp.com>; Donovan A. Schneider: Red Brick Systems, 485 Alberto Way, Los Gatos, California, 95032, USA, email: <donovan@redbrick.com>.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
©ACM

— *distributed file systems*; H.2.2 [**Database Management**]: Physical Design — *access methods*; H.2.2 [**Database Management**]: Systems — *distributed systems*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: data structures, distributed access methods, extensible hashing, linear hashing

1. INTRODUCTION

The general trend in computer applications is towards online processing. More and more applications are mission critical and require fast analysis of unpredictably large amounts of incoming data. The traditional architecture is to deal with data through a single processor and its main (RAM) memory with disk as secondary storage. Recent architectures attempt to benefit from distributed or parallel processing, using multiprocessor machines with a local RAM per processor, and/or distributed processing on a number of sites. Finally, client/server architectures are widely used to provide specialized services in shared servers.

The trend towards distribution and parallelism is due to several factors. The main one is that whatever capabilities a single processor or site could have, a pool of sites can provide more resources and power. A second factor is the existence of high speed links. The evolution is rather spectacular: 10 Mb/sec (Megabits per second) Ethernet links are common, 100 Mb/sec FDDI or TCNS are in mass production [Gallant 1992; Byte 1992] and 100 Mb-1Gb/sec links are coming, e.g., Ultranet, HIPPI, or Data Highways. Similar speed cannot be achieved using magnetic or optical disks. It becomes more efficient to use the RAM of another processor than to use a local disk. Clearly, distributed RAM files will allow a database system to perform operations that were not feasible in practice within the classical database architecture.

Furthermore, it is common to find in an organization hundreds, or even thousands of interconnected sites (processors), with dozens of megabytes (MB) of RAM per site, and even more of disk space. This allows for distributed RAM files reaching gigabytes (GB). Efficiently harnessing the power of these large distributed systems is an important open research problem [Vaskevitch 1994; DeWitt and Gray 1992]. The bottom line is that whatever the possibilities of a site or processor, distributed processing can offer much more [Tanenbaum 1995].

However, distributed processing should be applied wisely, and this may be difficult. A frequent problem is that, while the use of too many sites may deteriorate performance, the best number of sites to use is either unknown in advance or can evolve during processing. Given a client/server architecture, we are interested in methods for gracefully adjusting the number of servers, i.e., the number of sites or processors involved. We present a solution for the following context.

There are several *client sites (clients)* sharing a file F. The clients insert objects given OIDs (primary keys), search for objects (usually given OIDs), or delete objects. The nature of objects is unimportant here. F is stored on *server sites (servers)*. Clients and servers are whole machines that are nodes of a network, or processors with local RAM within a multiprocessor machine. A client can also

be a server. A client does not know about other clients. Each server provides a storage space for objects of F, called a *bucket*. A server can send objects to other servers. The number of objects incoming for storage is unpredictable, and can be much larger than what a bucket can accommodate. The number of interconnected servers can be large, e.g., 10–100,000. The pool can offer many gigabytes of RAM, perhaps terabytes, and even more of disk space. The problem is to find data structures that efficiently use the servers. We are interested in structures that meet the following constraints:

(1) A file expands to new servers gracefully, and only when servers already used are efficiently loaded.

(2) There is no master site that object address computations must go through, e.g., to access a centralized directory.

(3) The file access and maintenance primitives, e.g., search, insertion, split, etc., never require atomic updates to multiple clients.

Constraint (2) is useful for many reasons. In particular, the resulting data structure is potentially more efficient in terms of messages needed to manipulate it, and more reliable. The size of a centralized directory could be a problem for creating a very large distributed file. Constraint (3) is vital in a distributed environment as multiple, autonomous clients may never even be simultaneously available. We call a structure that meets these constraints an *Scalable Distributed Data Structure (SDDS)*. It is a new challenge to design an SDDS, as constraint (2) precludes classical data structures modified in a trivial way. For instance, an extendible hash file with the directory on one site and data on other sites is not an SDDS structure.

To make an SDDS efficient, one should minimize the messages exchanged through the network, while maximizing the load factor. Below we describe an SDDS called **LH***. LH* is a generalization of Linear Hashing (LH) [Litwin 1980]. LH, and its numerous variants, e.g. [Salzberg 1988; Samet 1989], were designed for a single site, or for a multiprocessor machine with shared memory [Severance, Pramanik, and Wolberg 1990]. LH* can accommodate any number of clients and servers, and allows the file to extend to any number of sites with the following properties:

—the file can grow to practically any size, with the load factor about constant, typically between 65–95% depending on file parameters,

—an insertion usually requires one message, three in the worst case,

—a retrieval of an object given its OID usually requires two messages, four in the worst case,

—a parallel operation on a file of M buckets, costs at most $2M + 1$ messages, and between 1 and $O(\log_2 M)$ rounds of messages.

This performance cannot be achieved by a distributed data structure using a centralized directory or a master site.

LH* is especially useful for very large files and/or files where the distribution of objects over several sites is advantageous for exploiting parallelism. A bucket of an LH* file can also be a whole centralized file, e.g., a disk LH file. It therefore becomes possible to create efficient scalable files that grow to sizes orders of magnitude larger than any single-site file could.

The basic algorithm for LH* was defined in [Litwin, Neimat and Schneider 1993]. This article describes the algorithm in more depth. First, it discusses more features — parallel queries in particular. It also reports more extensive performance studies that confirm the excellent performance for much larger files. The basic algorithm is also enhanced with an algorithm for load control. This results in a higher load factor than that of LH* without load control. The load control algorithms proposed for LH could not be applied to LH* because they would lead to a prohibitive messaging cost. An algorithm specific to LH* is reported that is based on estimating the load factor of the file on the basis of the load of a single bucket. While this principle naturally leads to some inaccuracy, its basic advantage is that the load control is obtained at no additional message cost. The algorithm is proved efficient through extensive simulations as reported in this article.

A basic property of LH* as described in [Litwin, Neimat and Schneider 1993] is that splits and merges are managed by a special site called the split coordinator (SC). The existence of an SC has advantages, but it also has drawbacks, e.g., the need for sending messages between buckets and the SC, and the vulnerability to failure. In this article, we define variants of LH* without any SC. Such variants require load control and some modifications to the splitting policy as defined in [Litwin, Neimat and Schneider 1993] because uncontrolled splitting could lead to unlimited file overload. The article proposes several strategies for splits in an LH* file without SC. They differ with respect to the conditions under which the next split should occur. The analysis shows the superiority of one of the strategies.

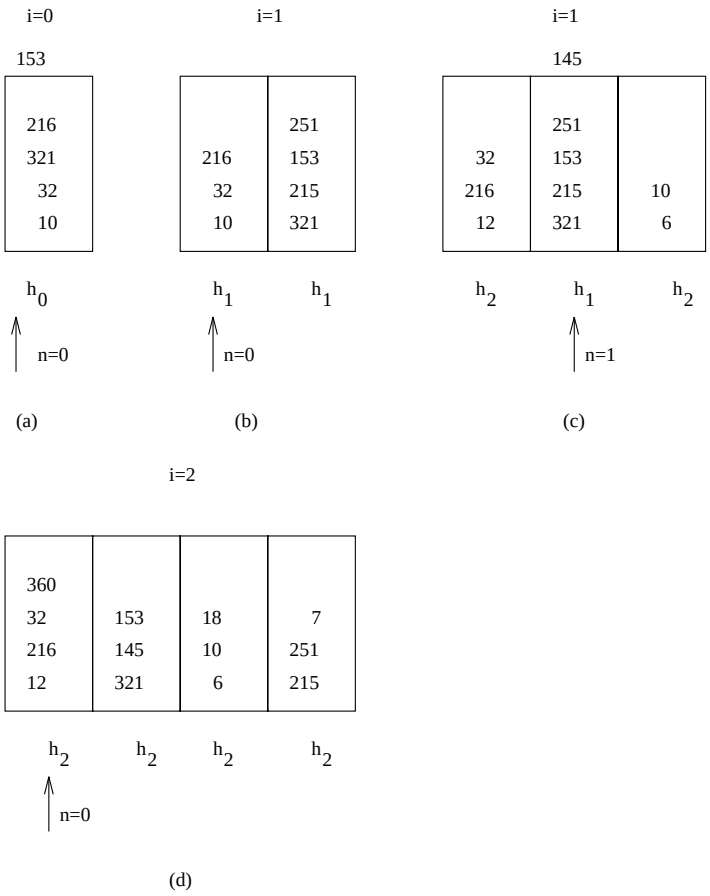
We also outline some variants of LH* that can be combined with any feature discussed earlier. One variant, called parallel splits, should increase throughput on fast networks, and under high insertion rates. Another variant, called presplitting, based on recursive LH [Ramamohanarao and Sacks-Davis 1984] may provide similar advantage, and allows for more autonomous splitting at each site. The drawback is that the strict bound on the worst case key search length does not hold. However, the expected deterioration should typically be negligible.

Section 2 contains a brief overview of Linear Hashing. Section 3 presents the basic LH* algorithm and in Section 4 its performance is analyzed. Section 5 describes several variants, including LH* without a coordinator, parallel splits and presplits. Related work is presented in Section 6. Section 7 concludes the article. A brief definition of common terms is included in Appendix A.

2. LINEAR HASHING

LH is a hashing method for extensible disk or RAM files that grow or shrink dynamically with no deterioration in space utilization or access time. We describe this method only briefly as the full description is widely available in the literature [Litwin 1980; Enbody and Du 1988; Salzberg 1988; Samet 1989] The files are organized into buckets (pages) on a disk [Litwin 1980], or in RAM [Larson 1988]. Basically, an LH file is a collection of buckets, addressable through a directoryless pair of hashing functions h_i and h_{i+1}; $i = 0, 1, 2...$ The function h_i hashes (primary) keys on $N * 2^i$ addresses; N being the initial number of buckets, $N \geq 1$. An example of such functions are the popular division modulo x functions, especially:

$$h_i(C) \rightarrow C \ mod \ N * 2^i$$



(a) original file w/153 causing a collision at bucket 0.

(b) after split of bucket 0 and inserts of 251 and 215.

(c) insert of 145 causes collision and split of bucket 0; 6 and 12 inserted.

(d) insert of 7 caused split of bucket 1; keys 360 and 18 inserted.

Fig. 1. Linear Hashing.

2.1 File expansion

Under insertions, the file gracefully expands through the splitting of one bucket at a time into two buckets. Figure 1 illustrates this process using functions h_i above, for $N = 1$ and a bucket capacity of 4. A function h_i is linearly replaced with h_{i+1} when existing bucket capacities are exceeded. A special value n, called pointer, is used to determine which function, h_i or h_{i+1}, should apply to a key (OID). The value of n grows one by one with the file expansion (more precisely it grows from 0 to $N-1$, then from 0 to $2N-1$, etc.). It indicates the next bucket to split and it is always the leftmost bucket with h_i.

A split is due to the replacement of h_i with h_{i+1}, and is done one bucket at a time. Typically, each split moves half of the objects in bucket n to a new address that is always $n + N * 2^i$. At some point, h_{i+1} replaces h_i for all current buckets. In this case, h_{i+2} is created, $i \leftarrow i + 1$, and the whole process continues for the new value of i. It can continue in practice indefinitely. The result is an almost constant access and memory load performance, regardless of the number of insertions. This property is unique to LH schemes. The access performance stays close to one disk access for successful and unsuccessful searches alike [Litwin 1980].

2.2 Split control

A split can be performed whenever a collision occurs. Such splits are called **uncontrolled**. The load factor is then about 65 – 69%, depending on the bucket capacity. Some applications, however, require a higher load. One way to achieve this goal is to monitor the load factor, and to perform a split not only when a collision occurs, but also when the load factor is above some threshold t. Such splits are called **controlled**. At the expense of a slight deterioration of access performance, the split control allows for a higher load factor, in practice equal to t. The threshold should be $t > 0.70$, it can even reach 95%.

2.3 Addressing

The LH algorithm for hashing a key C, i.e., computing the bucket address a to use for C, where $a = 0, 1, 2, ...;$ is as follows:

$a \leftarrow h_i(C);$ **(A1)**
if $a < n$ then $a \leftarrow h_{i+1}(C);$

The index i or $i + 1$ finally used for a bucket is called the *bucket level*. The value $i + 1$ is called the *file level*.

2.4 File contraction

Deletions in an LH file can trigger bucket merging. This operation is the inverse to bucket splitting. The pointer n moves backward by one address, i.e., $n \leftarrow n - 1$, and if $n < 0$ then $i \leftarrow i - 1$ and $n \leftarrow 2^i - 1$. Bucket n merges with the last bucket in the file and the space for the last bucket is freed.

The merging should occur when the load factor of the file becomes too low. Hence bucket merging requires some load control, e.g., through comparison to a threshold t'. If splits are not controlled, t' should be about 50–60%. In the case of load control, t' can be higher, even up to $t' = t$. This setting allows for the best stability of the load factor. However, it is rarely suitable in practice. It can lead to a continuous split/merge alternation concerning a few or even a single bucket. It is probably better to give up some load factor stability and allow for $t' < t$, by a few percentage points. A difference of 5-10% is reasonable.

3. OVERVIEW OF LH*

3.1 File expansion

We describe here the basic LH* scheme. For ease of discussion, each bucket is assumed at a different server (see Figure 2). Each bucket retains its bucket level (9 or 10 in Fig. 2) in its header. Buckets are in RAM, although they could be on

disk as well. The internal organization of a bucket is not of interest here as it is local to a site and implementation dependent. Overflows are basically assumed to be handled as for an LH bucket, e.g., chained in additional dynamically allocated storage as in [Litwin 1980; Larson 1988]. (An overflow and a collision occur when a bucket with capacity b contains b or more records and receives a request to insert a new record.) Buckets (and servers) are numbered $0, 1, ...$, where the number is the *bucket address*. These logical addresses are mapped to statically or dynamically allocated server addresses, as discussed in Section 3.5.

An LH* file expands as an LH file. Initially, the file consists of bucket 0 only, the pointer value is $n = 0$, and h_0 applies. Thus, addressing based on (A1) uses the values $n = 0$ and $i = 0$. When bucket 0 overflows, it splits, bucket 1 is created, and h_1 is used. Addressing through (A1) now uses the values $n = 0$ and $i = 1$. At the next collision, bucket 0 splits again, the pointer moves to bucket 1, and h_2 starts to be used, as shown previously. Addressing through (A1) now uses $n = 1$ and $i = 1$, etc. In Figure 2, the file has evolved such that $i = 9$ and $n = 80$. The last split of bucket 0 created bucket 512 and further splits expanded the LH* file to 592 servers.

3.2 Addressing

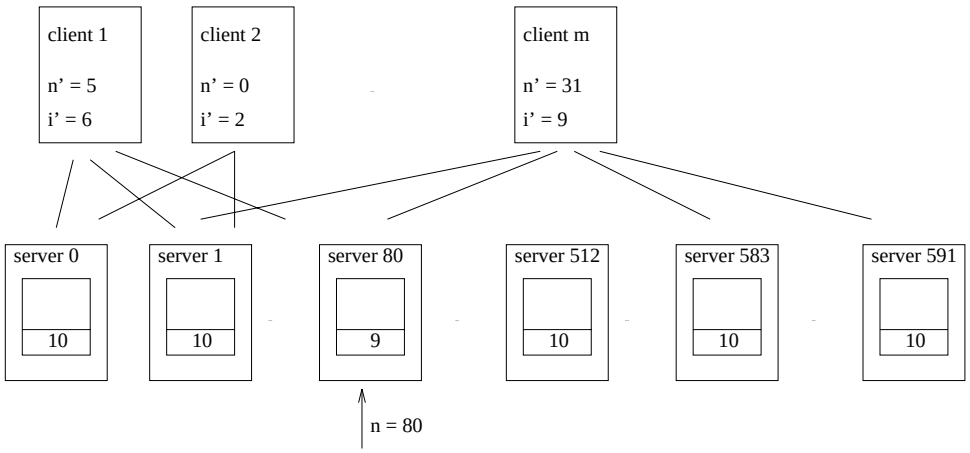


Fig. 2. Principle of LH*.

3.2.1 *Overview.* Objects of an LH* file are manipulated by clients. A client usually inserts an object identified with its key, or searches for a key. A client can also perform deletions or parallel searches as we address later. There can be any number of clients, as shown in Figure 2.

LH is based on the traditional assumption that all address computations use the correct values of i and n. Under SDDS constraints, this assumption cannot be satisfied when there are multiple clients. A master site is needed, or n and i values

need to be replicated. The latter choice implies that all clients should atomically receive a message with a new value of n with each split. Neither option is attractive.

LH* principles are different in that they do not require all clients to have a consistent view of i and n. The first principle is that every address calculation starts with a step called *client address calculation*. In this step, the client applies (A1) to its local parameters, n' and i', which are the client's view of n and i, but are not necessarily equal to the actual n and i of the file. The initial values are always $n' = 0$ and $i' = 0$; they are updated **only** after the client performs a manipulation. Thus, each client has its own *image* of the file that can differ from the file, and from images of other clients. The actual (global) n and i values are typically unknown to a client, they evolve through the action of all the clients. Even if n' equaled n the last time a client performed a manipulation, it could happen that splits occurred in the meantime and $n > n'$ or $i > i'$.

Figure 2 illustrates this principle. Each client has values of n' and i' it used for its previous access. The image of client 1 is defined through $i' = 6$ and $n' = 5$. For this client, the file has only 69 buckets. Client 2 perceives the file as even smaller, with only 4 buckets. Finally, client m sees a file with 543 buckets. None of these perceptions is accurate, as the actual file grew to 592 buckets.

A client inserting or retrieving a key C may calculate an address that is different from the actual one, i.e., the address that would result from using n and i. In Figure 2, each client calculates a different address for $C = 583$. Applying (A1) with its n' and i', client 1 finds $a = 7$, client 2 computes $a = 3$, and client m calculates $a = 71$. None of these addresses is the actual one, i.e., 583. The whole situation could not happen in an LH file.

A client may then make an *addressing error*, i.e., it may send a key to an incorrect bucket. Hence, the second principle of LH* is that every server performs its own *server address calculation*. A server receiving a key, first verifies whether its bucket should be the recipient. If not, the server calculates the new address and forwards the key there. If the file shrinks, a server may occasionally send a key to a bucket that does not exist any more; we delay the discussion of this case to Section 3.4. The recipient of the forwarded key checks again, and perhaps resends the key. We will show that the third recipient must be the final one. In other words, in the worst case, there are two forwarding messages, or three buckets visited.

Finally, the third principle of LH* is that the client that made an addressing error gets back an *image adjustment message (IAM)*. This message contains the level of the bucket the client first addressed, e.g., in Figure 2, buckets 0–79 and 512–591 are at level 10 while buckets 80–511 are at level 9. The client executes the *client adjustment algorithm* which updates n' and i', thus getting the client's image closer to the actual file. The typical result is that clients make very few addressing errors, and there are few forwarding messages, regardless of the evolution of the file. The cost of adjusting the image is negligible, the IAMs being infrequent and the adjustment algorithm fast.

We now describe these three steps of the LH* address calculation in detail.

3.2.2 *Client address calculation.* This is simply done using (A1) with n' and i' of the client. Let a' denote the resulting address.

$$a' \leftarrow h_{i'}(C); \qquad (A1')$$

if $a' < n'$ then $a' \leftarrow h_{i'+1}(C)$;

$(A1')$ can generate an incorrect address, i.e., a' might not equal the a that algorithm (A1) computes, but it can also generate the correct one, i.e. $a' = a$. Figure 3 illustrates both cases. The actual file is shown in Figure 3a with $i = 4$ and $n = 7$. Thus, buckets 0–6 are at level 5, buckets 7–15 are at level 4, and buckets 16–22 are at level 5. Two clients, Figures 3b–c, perceive the file as having $i' = 3$ and $n' = 3$. The client in Figure 3d has a still different image: $i' = 3$ and $n' = 4$.

Figure 3b illustrates the insertion of key $C = 7$. Despite the inaccurate image, the client sends the key to the right bucket, i.e., bucket 7, as (A1) would yield the same result. Hence, there is no adjusting message (IAM), and the client stays with the same image. In contrast, the insertion of 15 by the client in Figure 3c, leads to an addressing error, that is, $a' = 7$ while $a = 15$. A new image of the file results from the adjustment algorithm (explained in Section 3.2.4). Finally, the client in Figure 3d also makes an addressing error since it sends key 20 to bucket 4, while it should have gone to bucket 20. It ends up with yet another adjusted image.

3.2.3 *Server address calculation.* No address calculated by $(A1')$ can be beyond the file address space, as long as there was no bucket merging. (This assumption is relaxed in Section 3.4.) Thus, every key sent by a client of an LH* file is received by a server having a bucket of the file, although it can be an incorrect bucket.

To check whether it should be the actual recipient, each bucket in an LH* file retains its **level**, let it be j; $j = i$ or $j = i + 1$. In LH* files, values of n are unknown to servers so they cannot use $(A1)$. Instead, a server (with address a) recalculates C's address, noted below as a', through the following algorithm:

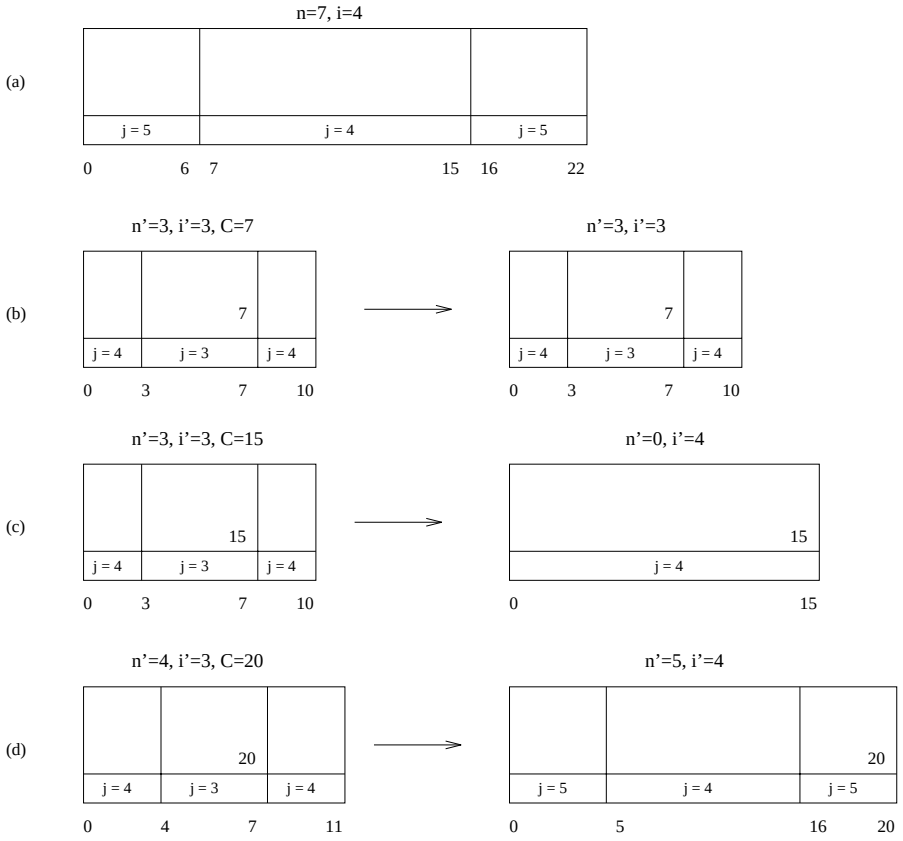
$a' \leftarrow h_j(C)$; **(A2)**
if $a' \neq a$ then
  $a'' \leftarrow h_{j-1}(C)$
  if $a'' > a$ and $a'' < a'$ then $a' \leftarrow a''$;

If the result is $a = a'$, then the server is the correct recipient and it performs the client's query. Otherwise, it forwards the query to bucket a'. Server a' reapplies (A2) using its local values of j and a. It can happen that C is resent again. But then, it has to be the last forwarding for C.

Indeed, consider that the file uses a typical network, e.g., Ethernet, under normal circumstances where messages are not unreasonably delayed [Abeysundara and Kamal 1991]. More specifically, assume that bucket a processes a forwarding message m_1, and then processes a split message m_2. Message m_1 should finish being processed at its target bucket before m_2 finishes being processed at bucket a (note that the processing of a split operation, following the reception of a split message requires in fact several messages, while the reception of a forwarding message using Algorithm A2 should be very fast). Key C is forwarded from a bucket a with level h_i or h_{i+1}. When it is processed at the target bucket a', this bucket still has level h_{i+1} at most. It may reach level h_{i+2}, only after all the buckets between and including a and a' undergo a split. This process should normally involve many messages and hence should be much longer than it takes to process one message. The following proposition holds then for LH* files:

PROPOSITION 3.1. *Algorithm (A2) finds the address of every key C sent through (A1′), and C is forwarded at most twice* [1].

The following examples facilitate the perception of the proposition, and of its proof, immediately afterwards. Note that the rationale in forwarding using $j-1$ is that the forwarding using j could send a key beyond the file.



(a) - actual file

(b) - inaccurate image, but no addressing error

(c,d) - image adjustments

Fig. 3. Images of an LH* File.

[1] Unless the LH* file has only a few buckets of very small capacity and the file undergoes heavy insertions [Vingralek, Breitbart, and Weikum 1994].

Example 1. Consider a client with $n' = 0$ and $i' = 0$, i.e. in the initial state, inserting key C where $C = 7$. Assume that the LH* file is as the LH file in Figure 1c with $n = 1$. Then, C is at first sent to bucket 0 (using $A1'$), as by the way, would any other key inserted by this client. The calculation using (A2) at bucket 0 yields initially $a' = 3$, which means that C should be resent. If it were resent to bucket $h_j(C)$, bucket 3, in our case, it would end up beyond the file. The calculation of a'' and the test through the second **if** statement prevents such a situation. It therefore sends key 7 to bucket 1. The calculation at bucket 1 leads to $a' = 1$, and the key is inserted there, as it should be according to (A1).

Assume now that $n = 0$ and $i = 2$ for this file, as shown in Figure 1d. Consider the same client and the same key. The client sends key 7 to bucket 0, where it is resent to bucket 1, as previously. However, the calculation 7 mod 4 at bucket 1 now yields $a' = 3$. The test of a'' leads to keeping the value of a' at 3, and the key is forwarded to bucket 3. Since the level of bucket 1 is 2, the level of bucket 3 must be 2 as well. The execution of (A2) at this bucket leads to $a' = 3$, and the key is inserted there. Again, this is the right address, as (A1) leads to the same result.

In the file in Figure 3, keys 15 and 20 are forwarded once. ∎

PROOF. (*Proposition 3.1.*)

Let a be the address of the bucket receiving C from the client. a is the actual address for C and there are no forwards iff $a = a' = h_j(C)$. Otherwise, let $a'' = h_{j-1}(C)$. Then, either (i) $n \leq a < 2^i$, or (ii) $a < n$ or $a \geq 2^i$. Let it be case (i), then $j = i$ (See Figure 4). It can happen that $a'' \neq a$, consider then that the

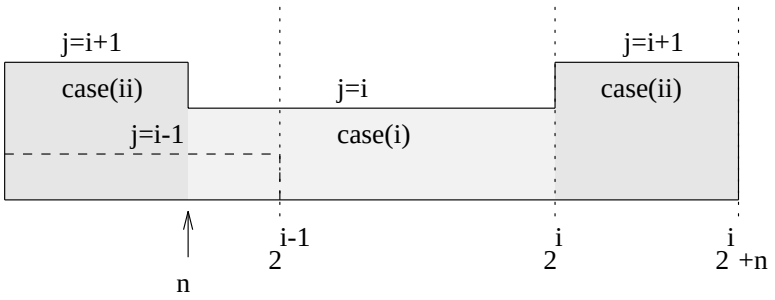


Fig. 4. Addressing regions.

forward to a'' occurs. If $a'' \neq a$, then, $i' < j - 1$, $a'' > a$, the level $j(a'')$ is $j = i$, and $a'' = h_{j(a'')-1}(C)$. Then, either $a'' = a' = h_i(C)$, or $a'' < a'$. In the former case a'' is the address for C, otherwise let us consider the forward to a'. Then, $j(a') = i$, and a' is the address of C. Hence, there are two forwards at most in case (i).

Let us now assume case (ii), so $j = i + 1$, and we must have $a'' \geq a$. If $a'' > a$, then C is forwarded to bucket a''. Then, $j(a'') = i$, or $j(a'') = i + 1$. In the latter case, $h_{j(a'')}(C) = a''$, so a'' is the address for C. Otherwise, $a'' = h_{j(a'')-1}(C)$, and it can happen that $a' = a''$, in which case a'' is the address for C. Otherwise, it

can only be that $a' > a''$, $a' \geq 2^i$, hence $j(a') = i + 1$, and a' is the address for C. Thus, C is forwarded at most twice in case (ii). □

(A2) implements the proof reasoning. The first and second lines of the algorithm check whether the current bucket is the address for C. The third and fourth lines trigger the forward to a'', if $a'' > a$.

3.2.4 *Client image adjustment.* In case of an addressing error by the client, one of the servers participating in the forwarding process sends back to the client an IAM containing the level j of the bucket a where the client sent the key. The client then updates i' and n'. The goal is to get i' and n' closer to i and n so as to maximize the number of keys for which (A1′) provides the correct bucket address. The LH* algorithm for updating i' and n' when an addressing error occurs is as follows. a is the address where key C was sent by the client, j is the level of the bucket at server a (j is included in the IAM).

1. $i' \leftarrow j - 1$, $n' \leftarrow a + 1$; **(A3)**
2. if $n' \geq 2^{i'}$ then $n' \leftarrow 0$, $i' \leftarrow i' + 1$;

Initially, $i' = 0$ and $n' = 0$ for each client. Figures 3c-d illustrate the evolution of images implied by (A3). After the image adjustment through Step 1, the client sees the file as with $n' = a + 1$ and with k buckets, $k \leq a$, with file level $j - 1$. Step 2 takes care of the pointer revolving back to zero. The whole guess can of course be inaccurate, as in both Figures 3c-d. However, the client view of the file gets closer to the true state of the file, thus resulting in fewer addressing errors. Furthermore, any new addressing errors result in the client's view getting closer to the true state of the file.

If all clients cease inserting objects, (A3) makes every n' converge to n. The convergence is not deterministic, but rather only probabilistic, i.e., the probability that eventually $n' = n$ increases with the number of searches. If there are insertions, then there are intermittent or even permanent gaps between n' and n, because of the evolution of n. A rarely active client makes more errors, as the gap is usually wider, and errors are more likely. Note that an intermittent gap exists during insertions even if there is a single client of the LH* file, unlike that of an LH file undergoing the same sequence of insertions.

Example 2. Consider Figure 3c. Before the adjustment, an addressing error could occur for every bucket in the client's image of the file, i.e., buckets 0–10, as for every such bucket the actual level was different from the considered one. The insertion of key 15 leads to a new perception — a file with level 4 for every bucket. This image differs from the actual file only at buckets 0–6. No addressing error can occur anymore for a key sent by the client to a bucket in the range 7–15. This should typically decrease the probability of addressing errors for this client.

For the client in Figure 3d, the insertion of key 20 leads to an image that is accurate everywhere but at two buckets: 5 and 6. Hence the probability of an addressing error becomes even smaller than in Figure 3c.

Consider that the client from Figure 3c subsequently searches for a key whose address is in the range 1–6. Every such search leading to an adjustment can only decrease the number of buckets with the level perceived as 4, instead of the actual level 5. The remaining buckets must be rightmost in the range. For instance, the

search for key 21 will lead to a new image, where only the level of bucket 6 remains incorrect. Under the uniform hashing assumption, the probability of an addressing error will become almost negligible (1/32 exactly). Finally, the insertion of a key such as 22 would make the image exact, unless insertions expanded the file further in the meantime. ∎

3.3 Splitting

As stated in Section 3.1, an LH* file expands as an LH file, through the linear movement of the pointer and splitting of each bucket n. The values of n and i can be maintained at a site that becomes the *split coordinator (SC)*, e.g., server 0. As for LH, the splitting can be *uncontrolled*, i.e., for each collision, or it can be *controlled*, i.e., performed for some but not all collisions.

3.3.1 *Uncontrolled splitting.* For uncontrolled splits, the split coordinator receives a message from each site that undergoes a collision. A collision message triggers the coordinator's message "you split" to site n, and triggers the LH calculation of new values for n and i by the SC using:

$n \leftarrow n + 1$; **(A4)**
if $n \geq 2^i$ then $n \leftarrow 0$, $i \leftarrow i + 1$;

Server n (with bucket level j) which receives the message to split:

(a) creates bucket $n + 2^j$ with level $j + 1$,
(b) splits bucket n applying h_{j+1} (qualifying objects are sent to bucket $n + 2^j$),
(c) updates $j \leftarrow j + 1$,
(d) commits the split to the coordinator.

Step (d) allows the SC to serialize the visibility of splits. This is necessary for the correctness of the image adjustment algorithm. If splits were visible to a client out of sequence, the client could compute a bucket address using an n that would be, for some period of time, beyond the file.

3.3.2 *Controlled splitting.* To control the splits as for LH, it would be necessary to send a message to the SC for every insertion and deletion. This would make the SC a hot-spot and would increase the number of messages, and hence is not acceptable. A practical split control algorithm should require only a few, possibly no additional messages. No such algorithm can therefore achieve the same stability of load factor as for LH. The following algorithm provides nevertheless efficient load control, as will be shown in Section 4. It has the interesting property of not requiring any additional messages between the buckets and SC. Several other strategies for load control are possible and are discussed in Section 5.

Load control strategy for LH.* Let bucket s be the bucket that undergoes the collision, b be the capacity of bucket s, where b is the same for all buckets in the file, x the number of objects in this bucket, and let d be $d = x/b$. Let t be some threshold given to SC, typically in the range $[0.7 - 1.0]$.

1. Bucket s sends a collision message to SC which includes x. **(A5)**
2. SC sets $d = x/b$ and then sets $d \leftarrow d * 2$, if $s < n$ or $s \geq 2^i$.

3. SC computes the formula: $\alpha' \leftarrow (2^i * d)/(2^i + n)$
4. If $\alpha' > t$, the SC causes bucket n to split. Otherwise no split takes place.

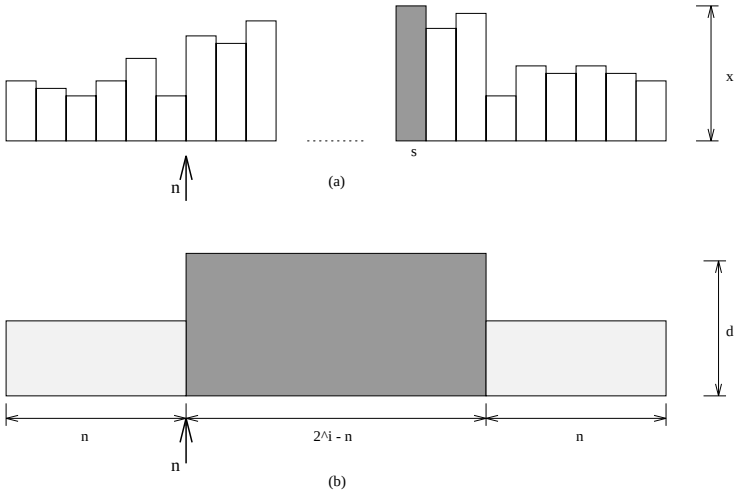


Fig. 5. File load. (a) actual (b) estimated.

Figure 5 illustrates the rationale for this strategy. If each h_i hashes uniformly then all buckets using the same functions are loaded to about the same level. As h_{i+1} hashes on twice as many addresses as h_i, one can further consider that buckets that already underwent a split with h_{i+1} are basically half loaded. In Step 2 and Step 3, the SC considers that every bucket's load factor is either (1) d or $d/2$ or (2) d or $2d$. It then attempts to maintain this estimate as close as possible to threshold t.

3.4 File contraction

The load control can allow an LH* file to shrink under deletions, as was possible for LH files. This can be done through a strategy similar to the one above. Namely, when the load estimate calculated by SC drops under some threshold, SC sends a message to bucket $n-1$ instructing it to merge with the corresponding bucket, and SC updates the value of n and i accordingly. As a result, the file has one fewer bucket, and the unused storage space is returned to the system.

As one should not generate messages to clients at each merge, a client of a shrinking file can address a bucket that does not exist anymore, i.e., with the address beyond the current file address space. To carry an insertion or a search to the right bucket, one solution is for the server to send an adjustment message to the client to downgrade its n' or i' in consequence. To completely avoid further addressing errors beyond the file, a sure bet is for the client to set n' and i' to zero, as bucket 0 is the last to disappear. Server 0 will then forward the key correctly

using (A2), and the client's image will be adjusted using (A3) as in the case of file expansion.

Other strategies may sometimes prove more efficient. For instance, if merges are relatively infrequent with respect to inserts and searches, then it should be more efficient to re-address the query to the address corresponding to $j-1$, where j is the level that the nonexistent bucket should have, i.e., $j = i'$ or $j = i' + 1$. This strategy would usually avoid the forwarding(s) of the first one. However, we will not address this subject in more depth. Shrinking files are seldom used in practice, and their addressing efficiency is not critical.

Finally, as already shown for LH files, and generally for any dynamic data structures based on bucket splitting and merging, it is preferable that the file not split and merge the last buckets cyclically under short sequences of inserts and deletes. To avoid this phenomenon, one should choose t' to be at least 10% less than t.

3.5 Allocation of sites

The bucket addresses a above are logical addresses to be translated to actual addresses s of the sites on which the file might expand. The translation should be performed in the same way by all the clients and servers of a file. The solution of a centralized name server being precluded, there are two approaches:

(1) the set of sites that files can use is known in advance and defined using a static table. This could be the case of a local net in a company, or of processors in a multiprocessor computer.

(2) the set of sites is defined for each file by a dynamic table that can become arbitrarily large, perhaps encompassing the entire INTERNET.

One solution for case (1) is to have at each site a table $T(a)$; $a = 0, 1, ..., M'$; with addresses of all available servers. Table T may actually be the addressing table of the networking system linking the sites. A few kilobyte table at each site should allow an LH* file to grow over hundreds or thousands of sites, i.e., all sites of a company or all processors in a supercomputer. Such a file can easily attain gigabytes in RAM, or terabytes on disk. If ever needed, T of a few megabytes should suffice for all INTERNET addresses, and a file of yet unimaginable size.

The most obvious strategy for assigning a site s to address a in a file is $s = T(a)$. However this strategy might not be the best for load balancing. The site $T(0)$ might become most loaded, since it is used by all the files. One may obtain better balancing through randomization. For instance, assuming that each file gets a unique number F when created, e.g., a time stamp, the value $F + a$ may be used as a basis for some hashing h over $[0, M']$, i.e., $s(a) = T(h(F + a))$. If some servers can handle more buckets than others, then more elaborate strategies may prove preferable.

One advantage of case (2), i.e., of a dynamic table, is better management of the space for the table. A more important advantage, though, is that sites for new buckets can be chosen dynamically and autonomously by buckets undergoing splits, e.g., through a contract net protocol. One simple solution for case (2) is then as follows.

A server informed to split creates a new bucket on the site of its choice. After performing the split, it sends the new address to the coordinator. Then, when a

client is notified of an addressing error with an image adjustment message, it sends i' and n' to the coordinator, which replies with the addresses not known to the client.

The coordinator will not become a hot spot, since, as will be shown, the number of addressing errors is typically small. Furthermore, the client is not blocked while waiting for the new addresses, it simply uses its existing image. The coordinator will need storage for all the addresses. Every other server only needs storage for the addresses of buckets it has created, i.e., for j addresses at most.

It is also possible to solve case (2) without the messages between clients and the coordinator, and in fact without any additional messages. Consider again that every bucket that splits sends the address of the new bucket to the coordinator. The coordinator then sends (within the split message to bucket n) the addresses that were created since the last split of bucket n. These addresses will be of buckets $[n + 2^{j-2} + 1,\ n + 2^{j-1}]$. The latter address will be chosen by bucket n, and sent later to the coordinator. Bucket n sends all of its addresses to bucket $n+2^{j-1}$ when it is being created. On the other hand, every bucket sends all the addresses it has within each image adjustment message. The price of this solution with respect to the previous one is more space needed at each server, i.e., for $n + 2^{j-1}$ addresses, and also longer image adjustment messages. However, it should usually be a minor problem in practice.

It can be observed that with this approach, the client may receive many addresses it already has. To send only the addresses new for the client, the server should know i' and n'. These values should then be enclosed within the the initial message to the servers, or the server can prompt the client for these values in the image adjustment message.

3.6 Parallel queries

A parallel query Q to an LH* file F is an operation which is sent by a client in parallel to selected, or all buckets of F, and whose executions at each bucket are mutually independent. Examples of parallel queries include a selection of objects from some or from every bucket according to some predicate, an update of such objects, a search for equal keys in every bucket to perform a parallel hash equijoin, a range search in every bucket, etc. A parallel query to selected buckets may result from some previous step, eliminating some specific buckets, for example, during an ordered traversal of the file. Parallel queries to all the buckets are more typical, and we restrict our attention to these queries, unless we state otherwise.

An LH* bucket replies to Q only if Q requests so. By default, the only reply is objects found if Q requests a search. Hence a parallel search may trigger between zero and as many messages as there are buckets in the file. Q may on the other hand ask for a reply from every bucket, e.g., even if there is no object found, to be sure that all buckets completed the search, or for an acknowledgement of every update operation. Such queries may be useful, e.g., in unreliable environments or for special operations, e.g., commitment, but should be rare, and are basically not considered in more depth in what follows.

A particularity of LH* is that a client might not know the extent of the file. Rather, it only knows the extent of its file image. If a bucket in an image was split after the last image adjustment, then the offspring of this bucket should get Q as

well. By the same token, Q should reach offspring of offspring, etc.

Support for parallel queries is a major advantage of LH* over LH. Similar operations on an LH file usually require sequential searches in the file that take orders of magnitude longer to complete. Two basic possibilities for sending out a parallel query are to broadcast (or multicast) the message, or to send a collection of point-to-point messages to the target buckets.

3.6.1 *Broadcast/Multicast messages.* The Ethernet protocol supports broadcast and multicast over sites on the same segment, as do many other popular network protocols [Abeysundara and Kamal 1991]. On an Ethernet segment, a broadcast (multicast) message takes the same amount of time to be delivered to all the controllers as does a point-to-point message. Ethernet can also support broadcast and multicast over sites on different segments, forming some local domain, provided the routers connecting the segments route such messages. This may not be the case, and is always limited to a well-delimited domain to avoid flooding too many sites or for security reasons. A broadcast message routed to another segment becomes another broadcast message.

The use of broadcast or multicast messages is possible for LH*. Although clients do not know the actual extent of the file, let it be F, every client of an LH* file may know the superset of sites with buckets of F. These are especially the sites with addresses in the static table T, dealt with in the previous subsection. The sites can be within a single segment, or on different segments of a local domain supporting broadcast and multicast. The multicast address can be preloaded for all and only sites in table T, as this set of sites is known in advance. The parallel query, let it be Q is then broadcast (multicast) at least to all sites in T. It is delivered to all LH* servers. Servers with buckets of F process Q, and send a reply to the client if the client so requested. The servers that do not have a bucket of F, and of course the sites where there is no LH* server at all, do not process Q nor do they send a reply.

The situation may be similar in the case of a dynamic table. It may be also different. If all of the servers in the dynamic table belong to the same broadcast domain, processing is performed as described above. If a file crosses multiple domains, the buckets whose splits crossed a domain must forward all broadcast messages.

It is to be noted that the use of multicast or broadcast, even if available, is not recommended for LH* with exact match queries unless the servers are underutilized. If the servers are already CPU bound, then each multicast/broadcast message will consume some of the CPU bandwidth of each server even if the server is not to handle the request. This will result in reduced throughput.

3.6.2 *Point-to-point messages.* LH* clients can use point-to-point messages if broadcast or multicast messages cannot be used. The client first sends Q using point-to-point messages to all the buckets in its image of F. As in the case of multicast messages, the client includes i' and n' in each message. Every bucket retains Q for processing and also forwards the message to each of its offspring beyond the client image. These offspring process the query, and forward the message to their offspring, if any, etc.

This strategy clearly allows the query to reach every bucket in F. The remaining problems are to guarantee (i) that every bucket gets the message only once, (ii) that

no buckets other than those in F get the message, and (iii) that for best performance the servers do not need to communicate with the coordinator. Requirement (iii) can clearly be satisfied if one uses an allocation algorithm as described in the previous subsection. The following algorithm for parallel query propagation at each server meets the other requirements.

LH parallel query propagation.* Consider that the client sent Q to every bucket in its image. Let bucket a be the bucket receiving a message containing Q. The initial values of a are all the addresses within the client's image. Every message also carries a variable called **message level**, noted j' . The initial value of j' in a message to bucket a is the level of a in the client's image, i.e., $j' = i' + 1$ for $a < n'$ or $a \geq 2^{i'}$, otherwise $j' = i'$. Bucket a (with file level j) then sends the query to each of its offspring using the following algorithm.

```
while j' < j do:                                              (A6)
    j' ← j' + 1;
    forward (Q, j') to bucket a + 2^(j'-1);
endwhile
```

PROOF. (*LH* parallel query propagation.*)
Every bucket a in the client's image receives Q from the client. The algorithm makes every bucket a forward Q in parallel in one round to all the offspring that it had created through its splits since the client's image was last adjusted. Every such offspring is beyond the client's image. The successive message levels are values of $i+1$ that had been used for these splits. Hence every offspring that was created by a bucket within the image gets Q, and gets it exactly once. The received message level is also the initial value of the level of the offspring that was created through the corresponding split. As every offspring that receives a forwarded message, and was split since the client's image was last adjusted, resends the message to all its offspring, etc., every bucket beyond the client's image gets the message, and gets it exactly once. As the last j' used for every bucket is $j' = j$, and for every bucket the level j is also $i+1$ of the last split function used, no message is sent to a bucket beyond F. Hence the algorithm meets the discussed requirements (i) and (ii). □

3.6.3 *Image adjustment.* The result of a parallel query Q can also be used for the client's image adjustment. It can even be done at no additional message cost. It suffices that Q lead to some replies, and that every corresponding reply message carry the actual level j of the bucket it came from. Let then j' be the highest j returned by a bucket replying to Q. And, let a' be the highest bucket address returning j' such that $a' < 2^{j'-1}$ and a''' be the highest bucket address returning j' such that $a''' \geq 2^{j'-1}$. Note that a' or a''' may be the empty set. The following algorithm executed by the client, if at least one result comes back, adjusts the image such that the extent of image is at most that of the actual file F, as for Algorithm (A3).

LH parallel query image adjustment algorithm*

```
1. i' ← j' − 1;                                               (A7)
2. If both a' and a''' are non-empty then n' ← max(a'+1, a'''+1−2^i');
   else if a' is non-empty then  n' ← a' + 1;
```

else if a''' is non-empty then $n' \leftarrow a''' + 1 - 2^{i'}$;

3. If $n' \geq 2^{i'}$ then $n' \leftarrow 0$; $i' \leftarrow i' + 1$;

PROOF. (*LH* parallel query image adjustment algorithm.*)

Addresses of results of Q returned to the client are seen here as a sampling of F's address space. The proof follows an examination of the different sampling cases.

(1) Consider that two different values of j are returned. j' is then set to the greater value $(i + 1)$, as defined above. Step 1 of algorithm A7 will then set i' to $j' - 1$, which is the correct value (i). Since at least one bucket returned a level of $j = i + 1$, a' or a''' (or both) must be non-empty. Step 2 examines all three cases and sets the value of n' accordingly. Since Step 2 can not set n' to a value greater than the true n, the condition in Step 3 will never be true.

(2) All buckets reply with the same j. The client can not determine the true level of the file since it can not determine whether the buckets are at level $j = i+1$ or $j = i$. Let us first consider the case where all buckets return level $j = i + 1$. In this case, Step 1 will set $i' = i$, which is the correct value. Step 2 will then set n' exactly as it did for case (1) where two different values of j were returned, and Step 3 will again have no effect. However, the case where all buckets return level $j = i$ is slightly more complicated. In this case, Step 1 will set $i' = i - 1$, which is one less than the true level of the file. a' and a''' will then be calculated using this value of i' and Step 2 will set n' accordingly. However, in this case n' may be set to a value beyond the end of the client's image (but still below the true value of the file). In this case, Step 3 will increase the client's file level by 1 and reset the pointer to 0. This new image must still be less than or equal to the true state of the file. □

How well the image gets adjusted depends obviously on the number of buckets that send back a reply. Assuming that addresses of these buckets are random, the more buckets reply the more likely it is that n' is set close to, or equal to n.

4. PERFORMANCE ANALYSIS OF LH*

In all the analysis in this section, and as usual for hash files, we assume that the hash functions used do hash the keys uniformly. In the case of LH*, it means specifically, as for LH, that with every h_i, the probability of hashing a key to any address is $1/2^i$ [Litwin 1980]. It is well known that distributions other than a uniform one may affect file performance [Knuth 1973], theoretically even hashing all the keys to a single address. The usual solution is to change the hash functions, e.g., as discussed for LH schemes in [Litwin 1980].

4.1 Basic features

The load factor x/bM, where x is the number of insertions, b is the bucket capacity in number of objects it can contain, and M is the current number of buckets, is the same as for LH when doing uncontrolled splits. It is thus around 65–70%.

The basic measure of access performance of LH* is the number of messages to complete a search or an insertion. A message here is a message to the networking system, we ignore the fact that it can result in several messages. For a random search for a value of C, assuming no address mismatch, two messages suffice (one to send C, and one to get back information associated with C). This is a minimum

for any method and is impossible to attain if a master directory site is necessary, since three messages are then needed. In the worst case for LH*, two additional forwarding messages are needed, i.e., a search needs at most four messages. We will see later in this section that the average case is around two messages and is hence better than any approach based on a master directory.

For a random insertion, the object reaches its bucket in one or, at most, three messages. Again, the best case is better than for a scheme with a master site, where two messages are needed. The bad cases should be infrequent, making the average performance close to one message. This is confirmed by simulation results in the next section. Globally, these values mean that LH* insertions are generally two times faster than for any scheme with a central directory[2]! In practice, a centralized scheme could also incur delays due to congestion at the directory site.

An insertion may also trigger a message to the split coordinator to initiate a split. This will occur on every bucket overflow. A split costs additional messages but these messages do not slow insertions. They can be performed asynchronously, are rare for a typical b, and the corresponding costs are marginal with respect to the averages.

The cost of initiating a parallel operation depends on the cases discussed in the previous section. If broadcast messages are available, then it may cost only one message to initiate a parallel operation if table T is used, or two messages, in two rounds, if it goes through the split coordinator. If only point-to-point messages are possible and assuming that M is the number of buckets in the file, then the cost is either $M + 1$ messages in two rounds, or M messages and between 1 and $i - i' + 2$ rounds. In the second case, the number of messages to initiate a parallel operation is the smallest possible, and is independent of the state of the client. The number of rounds is the smallest possible, i.e., one round of messages, when the client perception of the file is accurate, i.e., $i = i'$ and $n = n'$. Only less active clients require more rounds, but still the corresponding penalty is only $O(\log_2 M)$.

The cost of sending to a client the result of a parallel operation selecting x_p objects is $min(M, x_p)$ messages. Hence the total cost of a parallel operation is at most $2M + 1$ messages. We do not discuss parallel operations further in this paper.

Simulation modeling of LH*

We constructed a simulation model of LH* to gather performance results that were not amenable to direct analysis. Specifically, we wanted to ascertain performance in the average cases for file creation and searching, the expected load factor of files, the expected rate of convergence of a client view of a file, performance of rarely active clients, and the marginal costs of constructing a file. We show that average case performance is very close to the best case for insert and retrieve operations, that load factors can be effectively controlled, that clients incur few addressing errors before converging to the correct view of the file, that performance for very inactive clients is still quite good, and that file growth is a relatively smooth and inexpensive process.

We first describe the simulation model and then report the detailed results of the

[2]Even if insertions require an acknowledgment message to the client, LH* is expected to be 33% faster.

performance analysis. The simulator used the CSIM simulation package [Schwetman 1990].

The logical model of the simulator contains the following components. The *clients* model users of the LH* file which insert and retrieve keys, the *servers* each manage a single bucket of the file, the *split coordinator* controls the evolution of the file, and the *network manager* provides the intercommunication. More detailed behavior of each of these components is described below.

We assume a shared-nothing multiprocessor environment [Stonebraker 1986] where each node has a CPU and a large amount of local memory. Each server (and hence bucket) is mapped to a separate processing node, as is each client. The split coordinator shares the processor with bucket 0.

4.2 Simulation components

Clients: Clients typically act in three phases: a series of random keys are inserted into an empty file, the client's view of the file is cleared, i.e., i' and n' are set to 0, and finally, some randomly selected keys are retrieved.

In our implementation, a client may or may not receive an acknowledgement message for each insert command. If acknowledgements are required, a minimum of two messages is necessary to insert a key into a file — the original insert request from the client to a server and a status reply. If an addressing error occurs in the processing of such an insert, the adjustment message (IAM) is piggybacked onto the client reply. If acknowledgments are not required, a server sends the IAM directly to the client. In the case of retrieves, IAMs are always piggybacked onto the client reply. In any case, a client uses the information in the IAM to update its view of the file (using Algorithm A3 in Section 3.2.4).

Servers: Each bucket in an LH* file is managed by a distinct server. Servers execute algorithm (A2) to determine whether they should process an operation or forward it to a different server. If forwarding is required, an IAM is sent to the client unless it is possible to piggyback it onto the client reply (as explained above). Upon receipt of a **split** message from the split coordinator, a server sends an **init** message to create a new bucket of the file and then scans all the objects in its local bucket and transfers those that rehash to the new bucket. This step may require multiple messages if the number of objects exceeds the network packet size. However, in our simulations, bucket splits require a single message to transfer records during the split as we abstract away network-dependent details. We are also modeling a single-user environment where network traffic becomes irrelevant. When the transfer of objects is completed, a **commit** message is sent back to the coordinator to signify the completion of the bucket split.

Split coordinator: The split coordinator controls file evolution using either uncontrolled splitting or controlled splitting (see Section 3.3). The actual flow of messages required to split a bucket is shown in Figure 6. As shown by the figure, at least four messages are required for each bucket split. Furthermore, the coordinator only allows a single bucket at a time to be undergoing a split operation. All collision notification messages received from servers while a split is in progress are queued for later processing. In Section 5.2, we describe how to relax this constraint.

The split coordinator is the only centralized component in the system. This should not be a performance problem, though, because the traffic into and out of

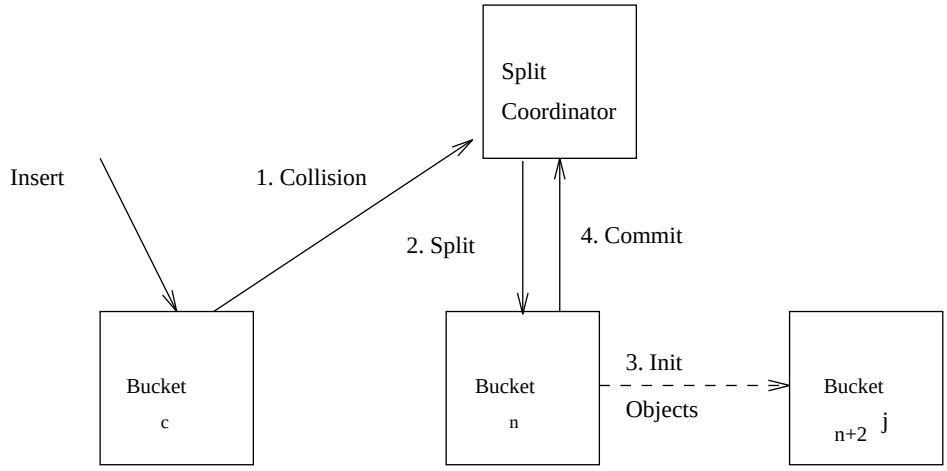


Fig. 6. Splitting of bucket n via split coordinator.

this entity is not significant, even if the file is growing rapidly. Thus, this component will not become a hot spot. For increased fault-tolerance, it would be relatively easy to replicate this activity. Nevertheless, we describe a variant in Section 5.1 that does not require a split coordinator.

Network manager: A common network interlinks the servers, clients, and the split coordinator. The network is restricted to one active transmission and uses a first-come, first-served (FCFS) protocol. Only point-to-point messages are supported.

4.3 Simulation results

In the simulation experiments, bucket capacities range from 50 to 10,000 objects. Buckets are managed in main-memory data structures. Clients insert from 10,000 to 1,000,000 randomly generated objects when building a file. Given the range of bucket capacities, this results in files where the number of buckets ranges from 20 to over 20,000. Clients typically retrieve 1,000 randomly selected objects.

4.3.1 *Performance of file creation and search.* Figure 7 shows the total number of messages per insert, the number of addressing errors per insert, and the number of messages associated with bucket splits required to build LH* files with bucket capacities ranging from 50 to 1000 objects. Three different files were constructed, with the number of objects varying from 10,000 to 1,000,000. Inserts did not require acknowledgements.

The leftmost graph in Figure 7 shows the total cost in messages per insert. That is, it includes the original insert messages from the client to the server, forwarding messages from server to server, IAMs from servers back to the client, and all the messages associated with bucket splits. The curves confirm our performance predictions in Section 3. Namely that performance is better for files with larger bucket capacities, and insert performance is very close to the best possible — one message per insertion. The figure shows that the difference is about 15% for small bucket

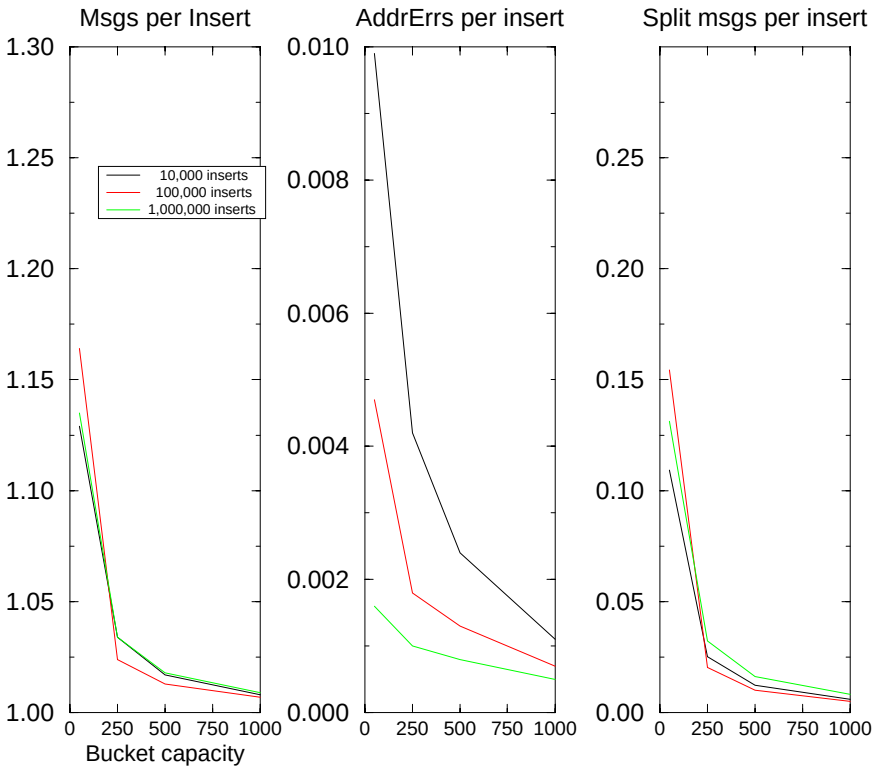


Fig. 7. Performance of file creation.

capacities and is under 5% for modest sized buckets, e.g., $b > 200$.

The middle graph in Figure 7 shows the relative contribution of messages associated with addressing errors (forwarding messages and IAMs) to the total number of messages. As is shown, these messages account for very little overhead; for larger bucket sizes the overhead is under 0.2%. Finally, the rightmost graph shows the contribution of messages associated with bucket splits. These curves show that split-related messages account for almost all the overhead associated with inserting objects. However, since bucket splits occur asynchronously and hence are not accounted for in the latency of an insert, from the point of view of the client the cost of an insert is only one message in practice.

Table I presents a more detailed picture of the curves in Figure 7 for 1,000,000 inserts and for bucket capacities ranging from 50 to 10,000. In addition to the average messages per insert (**Ave.Msgs**) it includes the number of addressing errors incurred for the inserts during file creation (**AddrErrs**). As is shown, the number of addressing errors is very small, even when the bucket capacity is small.

The column **Ave.Msgs-ack** in Table I shows the average number of messages per insert when the status of each insert operation has to be returned to the client. For example, this might be a requirement for clients that need stronger guarantees on the success of their updates. As is shown, these numbers are almost exactly one

Bucket Capacity	Build Perf.			Search Perf.	No. of
	AddrErrs	Ave.Msgs	Ave.Msgs-ack	Ave.Msgs	Buckets
50	1623	1.134	2.133	2.001	32791
250	1010	1.034	2.033	2.008	8070
500	771	1.018	2.017	2.008	4036
1000	558	1.009	2.009	2.008	2039
10000	78	1.001	2.001	2.006	128

Table I. File build and search performance (1000K inserts, 1K retrieves).

greater than the previous case where inserts are not acknowledged. The reason for being slightly less than one is that IAMs are piggybacked onto acknowledgement messages to the client.

The column **Search Perf** shows the average performance of a client retrieving random keys from the files. For each bucket capacity the client first inserted 1,000,000 random keys. It then reset its view of the file to empty and retrieved 1,000 keys. This entire process was repeated 30 times and the results were averaged. As the table shows, it requires just over two messages to retrieve an object, regardless of the capacity of the buckets. These values are very close to the best possible of two messages per retrieval, with the differences being under 1%.

Finally, it should be noted that this excellent performance is maintained for files with large numbers of buckets. For example, with a bucket capacity of 50, the file consisted of 32791 buckets, on the average.

4.3.2 *Performance of client image adjustment.* In this set of experiments we were interested in determining how efficiently a read-only client, starting with a view of the file as empty, obtains a true view of the file. That is, how efficient is the client image adjustment algorithm (A3)? Two metrics are of interest: the number of times the client sends the retrieval query to the wrong server (and hence made an addressing error and subsequently received an IAM) before its image converges to the true state of the file, and the total number of objects retrieved before convergence is reached. For each experiment, each data point was calculated by building a file with the required number of buckets, clearing the client's image of the file, and retrieving objects until the client image adjustment algorithm converges. This was repeated thirty times and the results were averaged.

These two metrics are shown graphically in Figure 8 for a file that grows from 128 to 1024 buckets, i.e., through three revolutions of the split pointer [3]. As expected, the lower curve shows that a client makes relatively few addressing errors before its view of the file converges to the true state. Also, as expected, the number of addressing errors increases slightly with the number of buckets. One may observe that the average number of addressing errors is between 1 and $\log_2$ of the number of buckets. This is intuitive, because, on average, upon receiving an IAM in response to an addressing error, the image adjustment algorithm halves the number of buckets that the client may address incorrectly. Although no formal calculus is known at present for values of n other than $n = 1$ or $n = 0$, these cases are clearly

[3] For this set of experiments, the results are independent of bucket capacity, but, note for the sake of completeness that b was set to 250.

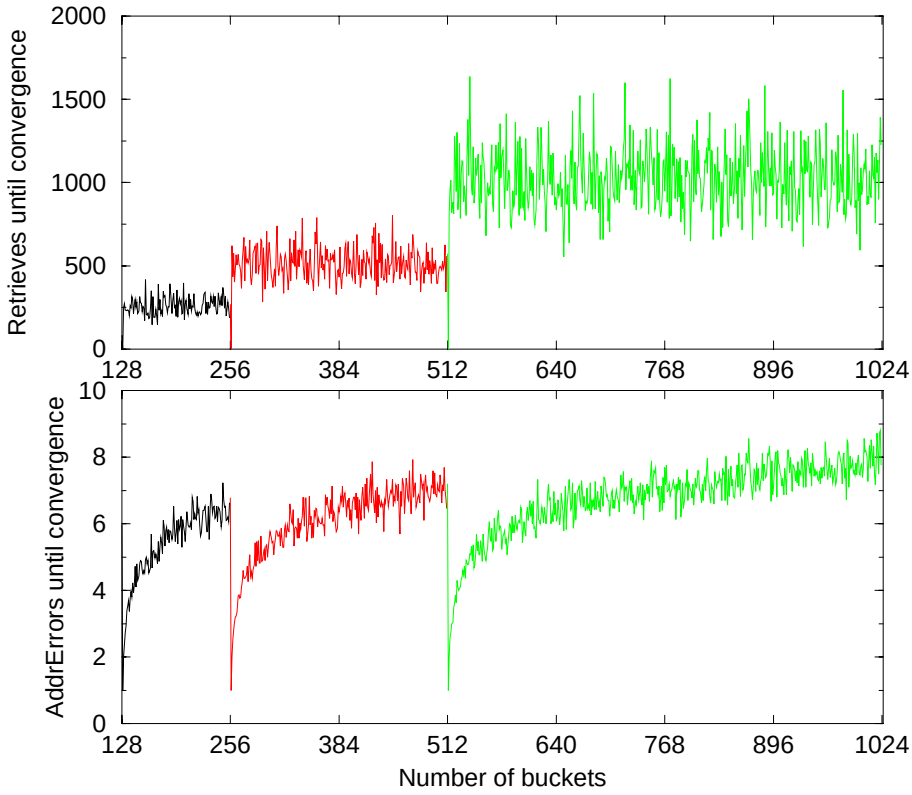


Fig. 8. Performance of client convergence algorithm.

the extreme values for the number of addressing errors.

The case of $n = 1$ is the best; in the figure this corresponds to the number of buckets being 129, 257 or 513. Whenever the pointer has this value, the file has buckets 0 and $M-1$ with level $i+1$ and all other buckets with level i. A new client with $i' = 0$ has to address bucket 0. Thus, the first IAM carries the value $i+1$. The image adjustment algorithm will then return the image that exactly matches the actual state of the file. Hence, any time $n = 1$, the client's image of the file converges to the actual state after the client makes its first addressing error. This is exactly what the figure shows.

The case of $n = 0$ is the worst case (number of buckets is 128, 256 or 512 in the figure). Let us consider that the file level is i, so the file has $M = 2^i$ buckets. Consider that h_i hashes uniformly. Then, all values of a are equally likely. On the average, the second error will occur on bucket $a = M/4$. Hence, n' will be set to $M/4+1$. After the adjustment, the next error will occur at one of the buckets in the range $[M/4+1, M/2-1]$. On the average, it will be the bucket $(3/8)M$ which will cause $n' = 3M/8+1$. By the same reasoning, the next error will occur on the bucket $7M/16$, etc. On the average, there will be at most $log_2 M$ addressing errors until the image reaches the actual file state. This is also what appears in the figure. The total number of addressing errors increases by one message every time the file

doubles. It remains therefore quite low.

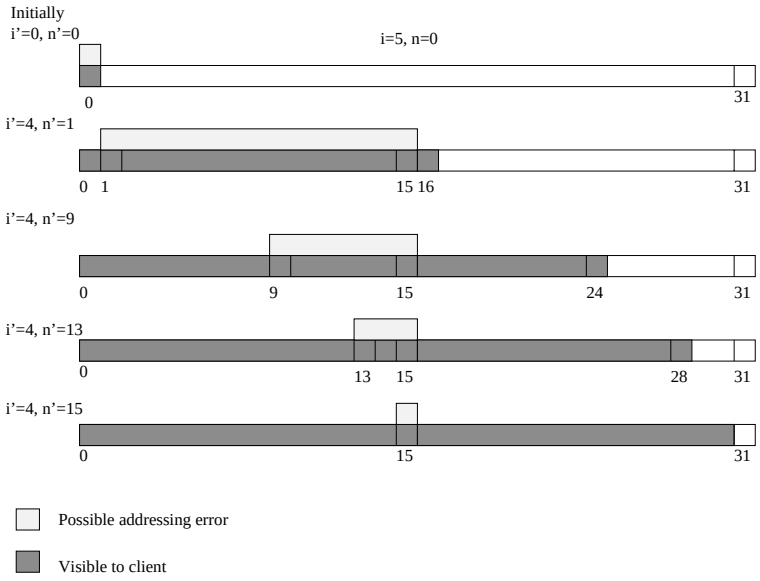


Fig. 9. A model of average convergence for a 32 bucket file.

As an example of the convergence process when $n = 0$, consider Figure 9. In this example, the file consists of $M = 32$ buckets, $i = 5$, and each server has $j = 5$. The client initially starts with a client image $i' = n' = 0$. Given its image, the client can only address bucket 0 and thus it can only make an addressing error to bucket 0. Upon making the first addressing error, the client updates its image to $i' = 4$ and $n' = 1$; it now "sees" the file as having 17 buckets. Given this image of the file, it can only make an addressing error at buckets 1–15. With the assumption of uniformity, the next error will occur at bucket 8. This will trigger the client to update its view to $i' = 4$ and $n' = 9$ and hence view the file as having 25 buckets. It can then make its next addressing error in the buckets 9–15, only. Assuming uniformity, this will be to bucket 12, thereby causing the client to set $i' = 4$ and $n' = 13$ and hence seeing a file of 29 buckets. The next addressing error will then occur for bucket 14. Finally, the next (and last) addressing error to bucket 15 triggers the client to update its image to $i' = 5$ and $n' = 0$ and thereby converges.

The intermediate values of n lead to results somewhere between the extremes. When n increases from $n = 1$, the portion of the file where addressing errors can occur also increases. For $n = 2$, the number will increase clearly to 2. For $n = 4$, it will be 3, etc. thus increasing towards the upper bound of $log_2 M$. Again this pattern can be observed on the figure, among the random noise.

The upper graph in Figure 8 demonstrates that a client retrieves many objects without incurring addressing errors even though the client's view of the file is inaccurate. For example, for the period where the file doubled from 512 buckets to 1024 buckets, the average number of objects retrieved was roughly 1000. Since the

average number of addressing errors for this same evolution of the file varied from approximately 6 to 8, it is clear that a client makes few addressing errors even when it has an imperfect view of the file.

These results hold for any bucket capacity. The performance of the client image adjustment algorithm is only dependent on the number of buckets in the file.

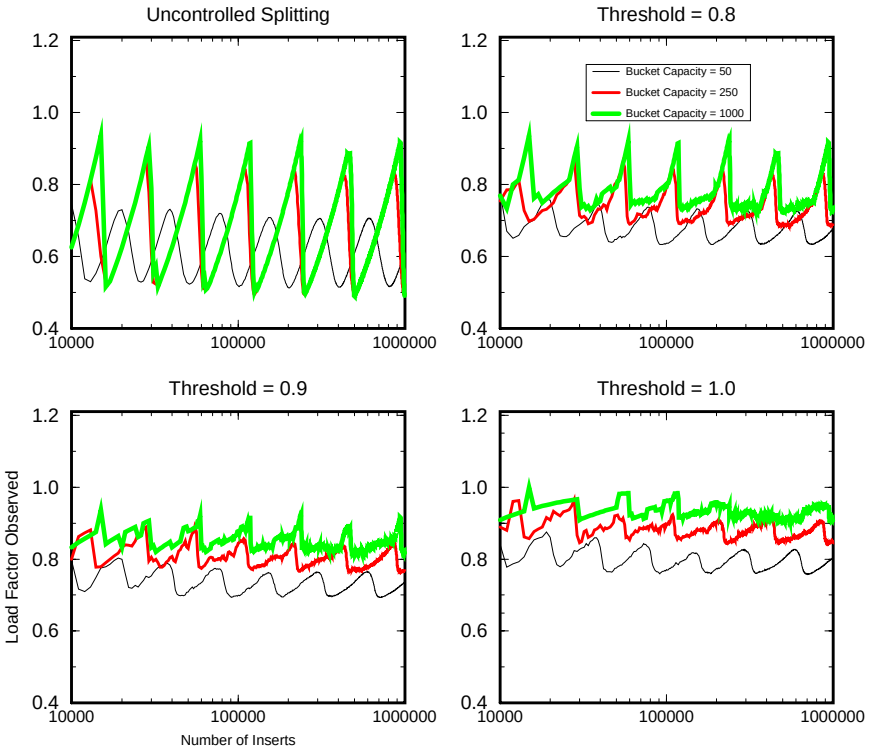


Fig. 10. Load control with split coordinator.

4.3.3 *Load factors of LH*.* Figure 10 shows the simulated evolution of load factor α in an LH* file undergoing up to 1,000,000 insertions and Figure 11 shows the same data grouped by bucket capacity rather than by threshold. The results are with load control and without load control (uncontrolled splitting), for various bucket capacities b and thresholds t. They demonstrate that an LH* file is typically efficiently loaded even without load control. If the load is controlled, it can further improve.

The correspondence between the actual load factor α, and the file parameters b and t is less straightforward than for LH. In LH, for t in the range of 0.7 to 0.9, the graph of α can simply be a horizontal line $\alpha = t$, i.e., the control can be tight enough to assure a constant load factor. The basic reason for the more variant shapes of α in LH* is that the coordinator only uses an estimate of the actual

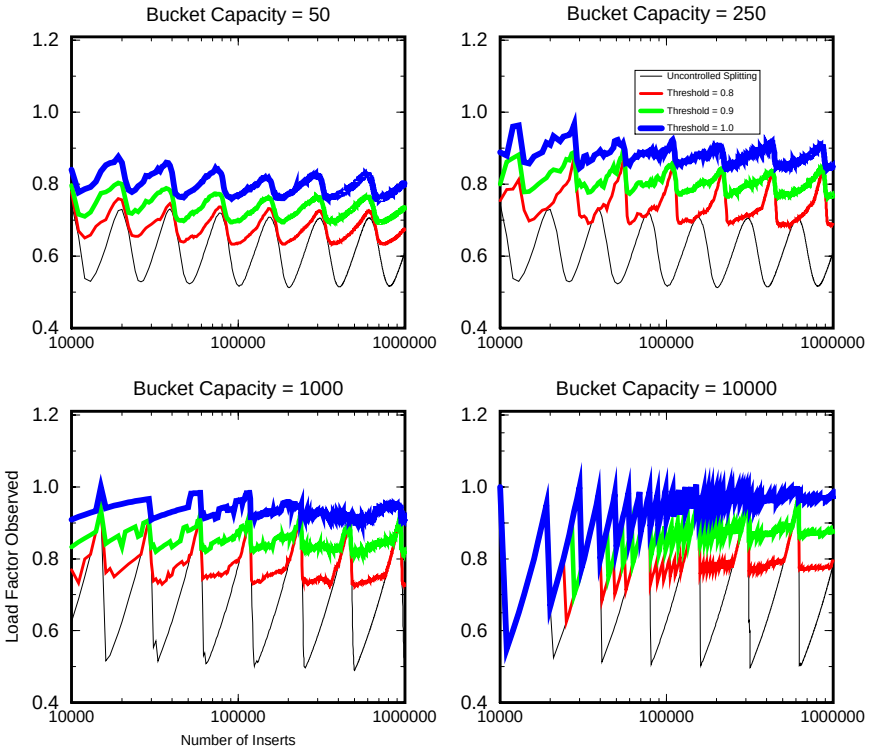


Fig. 11. Load control with split coordinator grouped by capacity.

load factor. For LH, the actual value of load factor can be computed. A similar approach in LH* would lead to an additional message per insertion, i.e., to a rarely practical approach.

Specifically, the curves lead to the following conclusions:

—The evolution of α without load control, is (of course) that of LH. Its characteristics are well known, both through simulations and analytical models, e.g., [Litwin 1980; Larson 1980]. The curves tend to be periodical in log scale. One oscillation corresponds to one full revolution of the pointer n, i.e., to the doubling of the file size. In the figure, the file increases 100 times, to 1,000,000 objects. This size is probably larger than most existing files. It is nevertheless also clear from the figures and confirmed by the analytical models, that the curves would remain similar for any file size, simply with more periods.

—The average value of every α (as measured over one period in stable state, i.e., after a larger number of insertions) is somewhere within 60–70% for uncontrolled splitting. This is typical of other schemes for dynamic files, e.g. LH, extensible hashing, or B-trees. The highest load occurs every time $n = 0$. The amplitude of a period increases with b, towards almost 50% for large b's. This pattern is also common. Larger buckets have a tendency to fill up simultaneously. The

probability of a collision is then close to one, leading to about simultaneous splitting by a few insertions, and to the drop of α to almost 50%.

—The curves corresponding to the load control show results for thresholds $t = 0.8$, 0.9 and 1.0. Every t improves α for every b in the studied range, as compared to uncontrolled splitting. For every number of insertions, and every curve, a higher t leads to a higher load. The average load factor and the minimum load factor improve therefore as well. For $b = 50$, $t = 0.8$ improves the minimal load from 52% to 63%, i.e., by about 10%, and the average from 62% to 68%. The threshold $t = 1.0$ leads to a minimum of about 75%, and to an average of almost 80%, i.e., to improvements of order 20%. For $b = 1000$, the choice of $t = 0.8$, moves the minimum of α from 50% to over 70%, and the average increases to about 75%. The choice of $t = 1.0$ moves the whole curve into the range 90–95%.

—In general, for a given b, increasing t leads to the minima increasing more than the maxima increases with respect to the uncontrolled splitting case. This is a desirable effect, as it improves the independence of α from the file state. The maxima move up more for smaller b's. For $b = 1000$ and $t \leq 0.9$ the part of every curve where $\alpha > t$, i.e., around each peak, coincides with the uncontrolled one. The load control thus cuts very effectively the underload of uncontrolled splitting, while it almost does not affect the file during the phases of the load higher than t. In particular, the higher t is, the lesser is the common part, as the load control starts blocking splits earlier (for smaller n's). In almost every case, the maxima remain under 100%. For large b's and $t = 1.0$, the whole phenomenon leads to an excellent almost constant load factor of about 90–95%, as for LH.

—Every choice of threshold in the range studied leads to the load factor usually under t, and lower than for LH. The load control appears also more effective for larger buckets as the average values and the minima are higher and closer to the threshold. Again, this is hardly a surprise. As for LH, it is also clear that the load factor will remain as in the figures, under any number of insertions into the file.

All things considered, the studied values of t appear to be practical. However, the exact correspondence between α, t, and b does not seem simple to assess. As long as there is no analytical model, to obtain the desired minimal and average load factor, it is probably best to guess the corresponding values of t and b from the simulation results in our figures.

4.3.4 *Performance of less active clients.* In this section, we analyze the performance of LH* when two clients are concurrently accessing a file. Specifically, we are interested in the case where one client is significantly less active than the other. The expectation is that a less active client makes more addressing errors than an active client since the file may evolve between accesses by the lazy client.

In these experiments, the two clients are synchronized such that the fast client inserts k objects for every object inserted by the slow client (k is referred to as the *Insert Ratio*). The fast client inserts either 100,000 or 1,000,000 objects. Thus, with an insert ratio of 100 to 1, the slow client only inserts 1,000 or 10,000 objects, respectively. Performance is reported as the number of messages required per insert for each client. Note that this metric does not include messages associated with

Insert	b=50, 100K ins.		b=1000, 100K ins.		b=50, 1M ins.		b=1000, 1M ins.	
Ratio	fast	slow	fast	slow	fast	slow	fast	slow
1:1	2.005	2.005	2.001	2.001	2.002	2.002	2.001	2.001
10:1	2.005	2.016	2.001	2.004	2.002	2.005	2.001	2.002
100:1	2.005	2.065	2.001	2.027	2.002	2.018	2.001	2.009
1000:1	2.005	2.200	2.001	2.140	2.002	2.058	2.001	2.046

Table II. Performance with two clients.

bucket splits since these messages typically take place in the "background" and hence a client does not directly "see" them. Also note that all inserts required an acknowledgement and hence the best case performance is two messages per insert.

The results are summarized in Table II for LH* files with a capacity of 50 and 1000 objects at each bucket and for insert ratios varying from 1:1 to 1000:1. As the table shows, for all experiments the performance of the slow client degrades as it is made progressively slower. This occurs because the inserts by the fast client expand the file thus causing the slow client's view of the file to become outdated. This then results in the less active client experiencing an increased number of addressing errors. However, it should be noted that performance is still quite acceptable for even extremely slow clients. For example, for an insert ratio of 1000 to 1 and small bucket capacities ($b = 50$), the slow client degraded less than 10% as compared to that of the fast client.

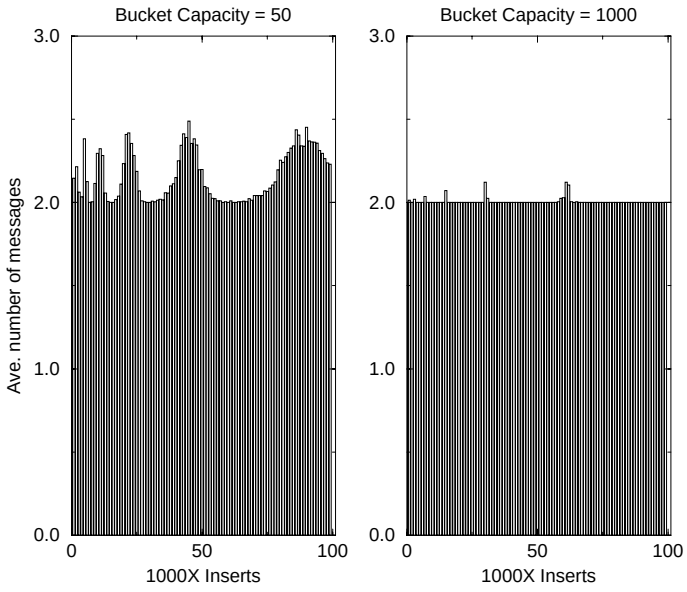


Fig. 12. Marginal costs of file creation using uncontrolled splitting.

4.3.5 *Analysis of marginal costs during file growth.* Many file access methods incur high costs at some points during file evolution. For example, in extendible hashing, the doubling of the directory is a very costly operation. In this set of experiments, we show that the performance of LH* files is relatively stable over their lifetime.

The experiments entailed populating LH* files with 100,000 random objects and measuring performance after every 1000 inserts. For example, the total number of messages required for each of the first 1000 inserts was gathered and averaged. The next batch of 1000 inserts was similarly measured. All inserts were acknowledged to the client and the the split coordinator used the uncontrolled splitting policy. Figure 12 shows the averages of each of the batch averages for the total message costs for LH* files with bucket capacities of 50 and 1000, respectively. As these graphs show, performance is relatively stable over the evolution of the files, especially for files with large bucket capacities. The small spikes that do occur are related to the splitting of buckets but, as the average number of messages stays between 2.0 and 2.5, client performance is not significantly affected.

5. VARIANTS OF LH*

In Section 3, the basic scheme for LH* was described. In this section, we describe two main variants. The first variant describes how to eliminate the split coordinator and the second describes how to perform splits out of order. Within the first variant, we describe, and evaluate through simulations, several strategies for controlling file load without a split coordinator. For the second variant we describe two separate means of performing splits out of order. These variants are not mutually exclusive.

5.1 LH* without the coordinator

The basic function of the coordinator in LH* is to decide when to split a bucket and which bucket is to be split. It also serves implicitly as the communication channel between the buckets that randomly undergo collisions, and those that linearly split. A potential drawback of having a coordinator is that it may fail. Other drawbacks are that every collision costs one message to the coordinator, and every split costs two messages to the coordinator. Thus, variants without a coordinator are attractive.

In this section, we describe schemes for performing uncontrolled splitting and controlled splitting without a coordinator. In all schemes, the value of the split pointer n becomes the current position of a **token**; $n = 0$ initially. When bucket n splits, it passes the token to the next bucket, that is bucket $n + 1$ or bucket 0, according to the LH pointer evolution scheme (bucket $(n + 1)$ *mod* 2^i to be exact). We assume here that bucket n always knows the physical address of the bucket to which the token should be forwarded. Only growing files are considered, the case of shrinking files is easy to infer.

We do not report the effect of dispensing with the coordinator on overall message cost. As seen from Figures 7 and 12, reporting a reduction in number of messages due to splits would have been hard to discern.

5.1.1 *Uncontrolled splitting.* With uncontrolled splitting, bucket n (the holder of the token) examines its load, let it be d, whenever it receives an insertion or when

it first receives the token. If $d > b$, i.e., a collision has occurred, the bucket splits, and forwards the token to the next bucket, which then becomes bucket n. This bucket similarly checks whether its load is above capacity, and, if so, splits and forwards the token to the next bucket. The splitting stops at the first bucket where $d \leq b$.

With respect to the splitting principles of LH and LH* described in Sections 2 and 3, respectively, this scheme introduces *cascading splits*. This feature is intended to prevent the file from becoming overloaded. Without cascading splits one can expect the file to be overloaded since the pointer does not move forward at every collision where $d > b$ as was the case with LH or previous versions of LH*. Instead, a collision on precisely bucket n, the token holder, has to occur. This is a much less frequent event. A cascade is necessary to clean overflows that have accumulated.

Note that it is even more important when there is no coordinator to properly choose the hashing functions. With a good hash function, it is very unlikely for insertions to avoid the bucket with the token for a long while. If such an unlikely event does occur, the file will get in a bad state. However, cascading splits would remedy the situation when the bucket overflows eventually. But, most importantly, the hashing functions should be replaced with functions that hash uniformly.

Performance of uncontrolled splitting. The necessity of split cascades is demonstrated in Figure 13. In this figure, 1,000,000 keys were inserted into files with bucket capacities ranging from 50 to 1000 records. The leftmost graph shows the file load resulting from a policy of no split cascades. Note that a y-axis value of 1.0 indicates that the file is loaded to 100% capacity. As the graph shows, the file is grossly overloaded, and the load is increasing rapidly for smaller bucket capacities. Even for larger buckets, e.g., b=1000, the file is sometimes overloaded and the load is growing, although it is hard to see given the scale of the figure. The rightmost graph shows the load of uncontrolled splitting with split cascades. As these curves show, the file is no longer grossly overloaded. However, the file is still overloaded for small buckets and the variance in the load is high. The load is even growing slightly with inserts despite the cascades.

This no-coordinator algorithm coupled with an uncontrolled splitting policy introduces a random delay in the splitting of bucket n, with respect to the same insertions and the corresponding coordinator-based algorithm. The evolution of the load factor is less smooth, i.e., with higher variance of load among buckets, and higher average overall load, than for the same policy with the coordinator (see Figure 10). The difference should be acceptable for large buckets, let us say $b > 250$, and uniform hashing. Indeed, the larger the bucket, the lower the variance in bucket load, for the same file level i. It is also more likely that if a collision occurred at some bucket a other than n, after relatively few insertions it will also occur at bucket n.

5.1.2 *Controlled splitting.* In Section 4, it was shown that a controlled splitting strategy, i.e., load control, significantly outperformed uncontrolled splitting with respect to the average load and the variance in the load. In this section, we present and evaluate several strategies for controlling the load without the use of a coordinator.

When the file is created, a threshold t is defined for load control. As will appear

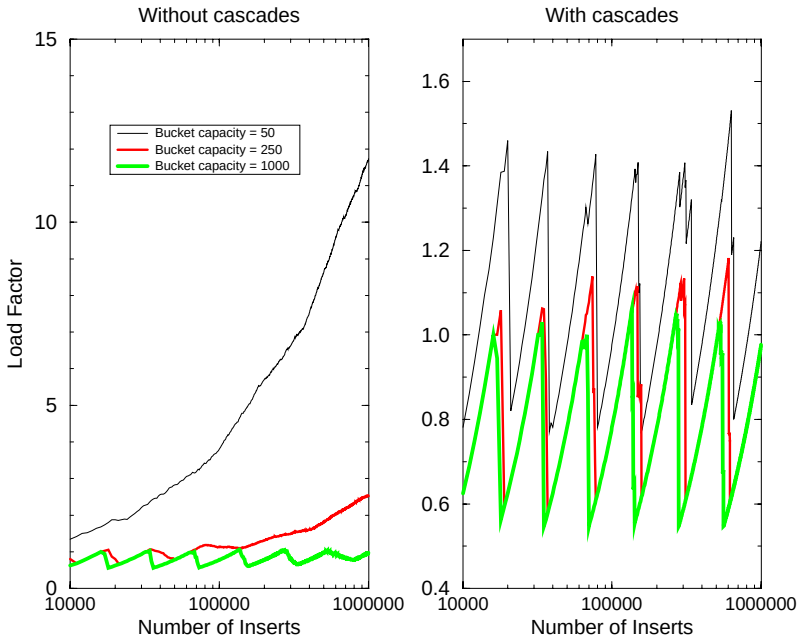


Fig. 13. File load for uncontrolled splitting—with and without split cascades.

later, one should set t to a value $0.7 < t < 1.0$. The basic scheme, let us call it **S1**, is that bucket n (the holder of the token) examines its load d whenever it receives an insertion or when it first receives the token. In the former case, the bucket tests whether $d > b$, i.e., a collision occurs. If so, the bucket inputs its load d to the estimate α' of the overall load factor that it computes using Algorithm A5 in Section 3.3.2, exactly as occurred in the case with the split coordinator. If $\alpha' > t$, the bucket splits and forwards the token to the next bucket. When a bucket receives the token, it immediately inputs its load to estimate α'. If it exceeds t, the bucket splits and forwards the token. The splitting stops at the bucket where the test of α' finds $\alpha' \leq t$.

Through the setting of $t < 1$, the load control has a capability to lower the load factor through cascading splits even if there is no collision at successive buckets. This capability will be justified soon. The lower the value of t, the longer the expected length of a cascade. Setting t close to 0.5 can clearly lead to long cascades, e.g., even all 2^i buckets may split in one pass. As each split lowers the load factor of a bucket, or creates an empty bucket, the length of every cascade is finite, provided of course $t > 0$.

On the other hand, observe that the average length of the sequence of insertions per bucket between successive splits increases with the file. This phenomenon did not exist for LH or for LH* with the coordinator. The file load factor can remain bounded, and under 100%, only if splits make sufficient room to absorb such a sequence of objects, on the average. This is one more rationale for introducing cascading splits, as a tool to increase the number of splits when the file grows. The

larger the file, the more likely it is that if a collision occurs at bucket n, it already occurred at the next bucket as well. This is also the rationale for the capability of the load control to lower the load factor through cascades and $t < 1$, i.e., even if there are no collisions at successive buckets.

In the basic scheme, S1, a cascade can only start when bucket n undergoes a collision. This tends to delay the triggering of a splitting cascade. It also means that by the time bucket n undergoes a collision, many of the buckets that follow it are ready to split as well. Thus, splitting cascades tend to be withheld longer than necessary, and once started, they tend to be long and several of them are likely to occur one after the other. Following these successive splitting cascades, the load of the file is likely to drop low enough so that splits do not occur for a while. This causes a wider variance in load than with the coordinator-based algorithm. An additional concern may be that longer and closely triggered cascades may saturate the network.

One solution is to limit the length of a cascade to at most a few buckets. This would avoid a sharp drop in the load factor, when n increases from $n = 0$, making its convergence towards t more amortized. Another way is to split the first bucket of a cascade earlier, i.e., when the estimate of file load reaches t, instead of waiting until the bucket load reaches b and the estimated file load reaches t. The corresponding strategy, which we call **S1′**, should allow for a more uniform (flat) load factor, with a value close to t, regardless of n.

On the other hand, one may observe that bucket load d acts in $S1$ as an estimate of an overall load of buckets that did not split yet during the current trip of the token. An underloaded bucket can stop splitting, leading to a longer cascade later. Also, the estimate is naturally less accurate for smaller buckets. A solution is to replace d with an average d' calculated over several recently split buckets. One may expect that the file evolution will be more smooth, with lower peaks of load factor and less variant length of cascades. One way to do it at no additional message cost is to piggyback the current length of the cascade and the current d' onto the token. The bucket receiving the token can then update d' using its own d value and decide whether to split. It does not seem worthwhile to keep averages from one cascade to another, as these values should typically be rapidly outdated.

The idea of averaging bucket loads leads to two more strategies corresponding to $S1$ and $S1'$ except for the use of d' instead of d. We call them $S2$ and $S2'$. As each scheme, while removing some drawback, also introduces some additional complexity, the basic question is when are the corresponding tradeoffs worthwhile in practice.

Performance analysis. Figure 14 shows simulations of file load factors for strategies S1, S1′, S2 and S2′ for bucket capacities b of 1000 and 50 and threshold $t = 0.8$. Each file underwent 1,000,000 insertions. As in Section 4.3, the keys used to build the file were randomly generated. The curves confirm the expectations. Specifically, for a large b, in this case $b = 1000$:

—The averaging does not have practical importance as S1 and S2 perform almost identically, and S1′ works about identically to S2′.

—Strategies S1 and S2 exhibit periodic peaks of load factor reaching 100%. The peaks correspond to $n = 0$. The peaks in load are resorbed after relatively few

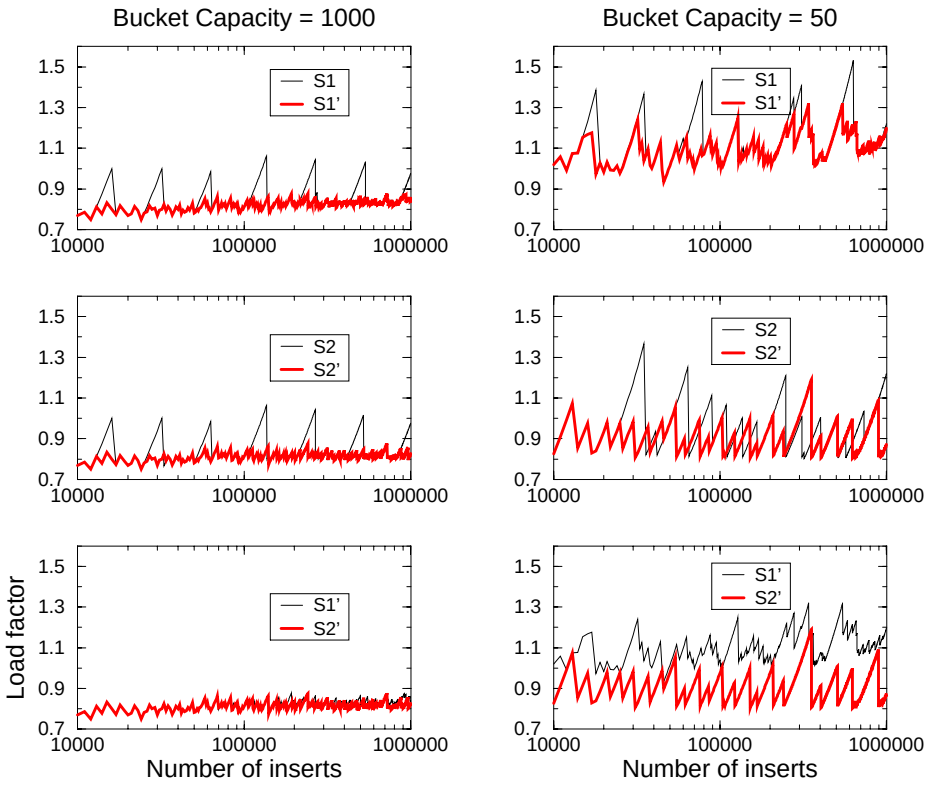


Fig. 14. Load control without split coordinator (b=1000 and b=50, t=0.8).

insertions, as was expected. Afterwards, the load factor remains within a few percent of the threshold value.

—Strategies S1′ and S2′ lead to a load factor that is within a few percent of the threshold value all the time.

—Neither the average, nor the peak load factor visibly grows with insertions. The curve for S1 can give an impression of slight growth, but a closer look on the right half shows that it is on the average flat (gradient is zero). This half corresponds in fact to 90% of the insertions because of the logarithmic scale.

For smaller b values, in this case $b = 50$:

—The curves present visibly higher variance, as expected. They are also in general above the threshold. This is true for both peak values that now can reach 150% and for average values that are between 90% and 110%.

—The averaging is more effective. Strategies S2 and S2′ are clearly better than S1 and S1′. Peaks of S1 reach 130–150%, while S2 is typically under 130%. Similarly, S1′ oscillates between 90 and 130%, typically around 110%, while S2′ is typically between 80 and 100%.

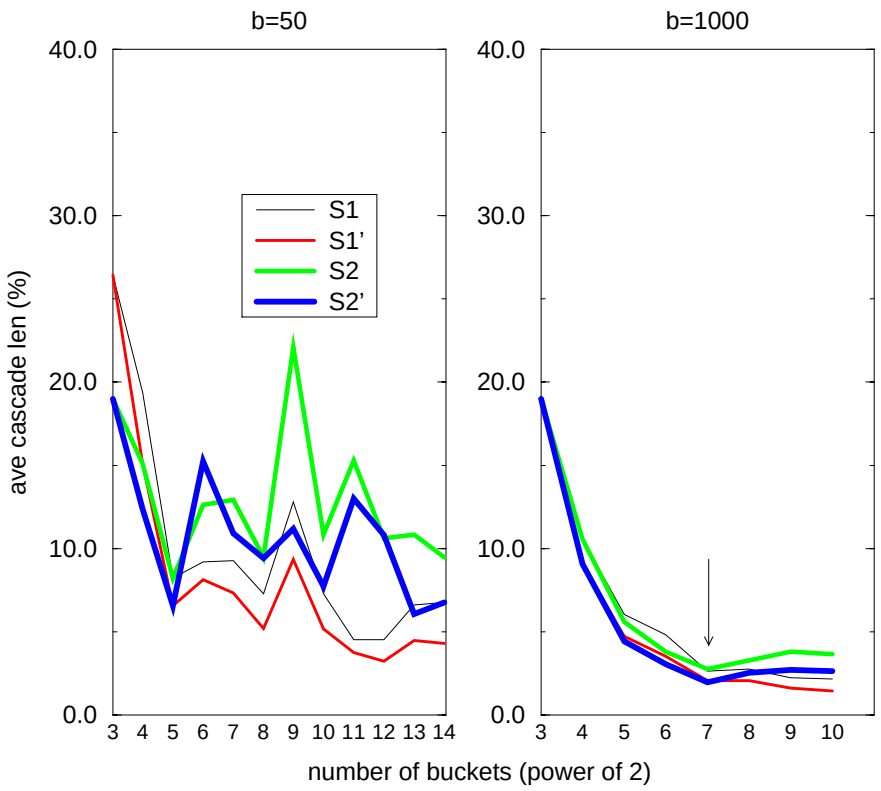


Fig. 15. Length of cascades (b=50 and b=1000, t=0.8).

Figure 15 shows data on cascading splits corresponding to the curves in Figure 14. The x-axis shows the number of buckets in the file (in powers of 2). The y-axis shows the average cascade length expressed as a percentage of the current file size. The average cascade length is calculated for each doubling of the file and plotted. For example, the data point for S2′ marked in the figure (y=2.5%, x=7) means that, on the average, each cascade covered 2.5% of the file as the file doubled from 2^6 to 2^7 buckets.

The results in Figure 15 match the intuition. A comparison of the two graphs shows that cascades tend to be longer for smaller buckets (b=50). For such buckets, the cascades are longer for strategies S1 and S2, which wait until an overflow has occurred before splitting, as compared to S1′ and S2′, respectively, which start splitting when the threshold is reached. Furthermore, they are longer for the strategies S2 and S2′ that average the bucket load to decide on further splits. Whenever a cascade occurs, several splits are performed, instead of always a single split at a time for schemes with the coordinator. However, as the figures show, a cascade affects a small fraction of the file for all the strategies, especially for larger buckets.

It is also interesting to compare the file load for S2′ to those in Figure 10 ($t = 0.8$) for the system with load control using a split coordinator. For small buckets, e.g., $b = 50$, the maximum load using the SC was always under the threshold, with an

average load of approximately 70%. Without the split coordinator, and for the same bucket capacity, the minimum load for S2′ was at the threshold, with the average around 90% and the maximum exceeding 110% at times. The results for S2′ are even better for larger bucket capacities. At $b = 1000$, the file load varies very little from the threshold. In contrast, the load with the SC is much more variant, with the maximum reaching 100% and the minimum dropping below the threshold. Thus, overall the file load can be controlled as well as, or better, by an algorithm that does not require a split coordinator. This surprising result is explained by the fact that S2′ utilizes split cascades and load averaging for maintaining file load.

In conclusion, the performance analysis shows that for larger buckets all four strategies perform about equally well. For small buckets, strategies S2 and S2′ are preferable. All things considered, S2′ appears the most practical.

5.2 Concurrent splits

For all variants of LH* we have discussed, it was assumed that the next split starts only when the previous one terminates. Such sequential splitting guarantees that no client that gets an error message from bucket m, sends a key to a bucket that should be created by a split of bucket $m-k$, but was not. The client getting j from m, will indeed infer that the level of bucket $m-k$ is also j. It could then send a key to bucket $m-k+2^{j-1}$. This bucket must exist if every split of a bucket $m-k$ must be terminated before the split of bucket m.

A coordinator using sequential splitting does not send a split message to bucket $n+1$ as long as it has not received a message from bucket n informing it that the split has finished. Similarly, in the variant without a coordinator, bucket n does not forward the token to bucket $n+1$ until it finishes its split. It may nevertheless be faster to perform splits concurrently, i.e., to start the next split without waiting for the previous one to finish. In theory, the time to perform k concurrent splits can be reduced by a factor of k. In practice, this is only possible if the network has sufficient bandwidth.

It is possible to enhance LH* for concurrent splits. Assume that the file has a *committed split pointer (token)* in addition to the regular split pointer (token), considered up to now. The regular pointer is used as before to determine the next bucket to split. The committed split pointer trails the regular pointer. When bucket n starts splitting, the regular pointer moves forward, i.e., the n value is set to $n \leftarrow n+1$. The regular split pointer moves to the next bucket any time a new split is started. The other pointer moves forward only when a split is finished, making the split *committed*. A client is informed of the new level $i+1$ (through IAMs) of the bucket only for committed splits. Buckets with splits yet uncommitted send out the old level i. Hence, no client can get $j = i+1$ if it could get $j < i+1$ for a previous bucket. Whatever the client's adjusted image is, all buckets in the image must exist, as for basic LH*.

For a scheme with a coordinator, the existence of two pointers means that the bucket that got a split message has to get an additional commit message. The message can be sent by the coordinator, or directly by the previous bucket. Similarly, in the variant without the coordinator, there has to be two tokens, and also one more message per split. The price for faster splitting is thus a slight slow down of every split. Thus, the insertion rate has to be high enough to make concurrent

splitting worthwhile.

What such conditions exactly are remain to be determined. In general, however, it is easy to see that concurrent splitting is naturally more interesting for variants without coordinators using cascading splits. It is particularly worthwhile then to use an allocation strategy putting buckets n and $n+1$ on different segments of a network. Concurrent splitting can be interesting for buckets on the same segment when the CPU time to perform a split dominates the corresponding network transfer time, and high insertion rates should be achieved. If the network is comparatively much slower or lacks sufficient bandwidth, concurrent splitting on the same segment may not be effective. Not only may throughput not increase, but it may decrease. On a typical 10 Mb/sec Ethernet more collisions may occur, leading to retransmission of some packets at best, and to network thrashing at worst.

Presplitting. The idea in concurrent splits is that the split of bucket $n + k$ can start before that of bucket n has finished. One can extend it further, assuming that a split of any bucket m can start and finish before it becomes bucket n. The splits are only committed in sequence of the movement of split pointer n, to avoid the problem discussed previously for concurrent splits. This principle is called *presplitting*.

Presplitting works as follows. For simplicity, we assume the existence of the coordinator, as the extrapolation to the variant without it is straightforward. Every bucket m has two levels, noted j and j', with $j' = j$ if the bucket has not presplit. Assume that bucket m, where $m \neq n$, with $j' = j$ overflows. As usual, it notifies the coordinator of the collision. However, it now sets $j' \leftarrow j' + 1$ and splits using $h_{j'}$, despite the fact that $m \neq n$, i.e., it is not its turn to split. A new bucket m' is created, exactly as it would be created if m was $m = n$ and the split using h_{j+1} was used. Levels j and j' of bucket m' are set to the updated value of j' of bucket m. The whole operation is a *presplit*.

From this point on, i.e., as long as $j' > j$, bucket m notifies the coordinator that a collision occurred for every insert it gets, although usually it still has excess capacity since half of its keys moved to bucket m' during the presplit. This policy allows the coordinator to keep the same pace of the pointer movement as it would have without presplits. If bucket m experiences a true collision, it again presplits. A collision can also occur on bucket m', triggering a presplit. In either case, the new bucket m'' is created with levels j and j' set to the updated level of j' of the presplitting bucket. When the pointer n comes to a presplit bucket m, the bucket simply sets j to $j \leftarrow j + 1$. If $j = j'$, then the split has "caught up" with the corresponding presplit(s) and the bucket no longer reports insertions as collisions until its capacity is again exceeded.

The addressing principles of LH* are modified as follows. If bucket m gets a key c and finds that it should be the addressee (using j in algorithm A2), it still needs to check j'. If $j' > j$, bucket m recomputes the address $h_i(c)$ using successively $i = j + 1, j + 2, ...$ until either $h_i(c) > m$ or $i > j'$. The former case means that c should be forwarded because of presplitting. The bucket resends c accordingly, and *tags* the message as being a forwarding due to a presplit. The recipient performs a similar address calculation, and either retains c or forwards it again. The forwarding process must obviously terminate. The final bucket processes the query, but

given the tag, it does not send an IAM to the client. Hence, presplitting remains transparent to the client.

Presplitting without a coordinator and without load control works similarly. In contrast, load control, with and without a coordinator, has to be rethought if presplitting is used. The problem arises because presplitting allocates buckets without the knowledge of the coordinator, or of bucket n with the token, making the corresponding load estimate void. One way to control the load is to hold a presplit until bucket m overflows to some level. This level would be a function of j and m. The definition of this function is an open problem.

Presplits should reduce the time and messaging cost to perform a cascade, as many of the buckets may already have presplit. On the other hand, the presence of presplit buckets will likely deteriorate access performance. More forwarding messages per search or insert may be needed, and the corresponding number becomes unbound in the worst case.

However, the deterioration in access performance should not be a problem in practice. On the one hand, strategies could be designed to accelerate the pointer (or token) in order to prevent too many presplits. Such strategies could be similar to those for load control, although with the goal of controlling access performance. On the other hand, one can observe that presplitting borrows many ideas from Recursive Linear Hashing [Ramamohanarao and Sacks-Davis 1984], which in turn borrows the idea of immediate splitting of overflowing buckets from extendible hashing [Fagin et al. 1979]. The performance analysis in [Ramamohanarao and Sacks-Davis 1984] makes likely the conjecture that the average access performance should deteriorate only slightly, even without load control. The performance analysis specific to presplitting remains to be done.

6. RELATED WORK

In a distributed file system (DFS), the files should reside at different sites, and the distribution of the files, with respect to remote access and storage, should be transparent to the users [Levy and Silberschatz 1990]. Various approaches, categorized in Figure 16 have been proposed towards this goal. In most DFSs, each file resides entirely on a site. Historically, this idea was the first to appear, and it is currently present in major commercial systems, NFS, and AFS in particular. Its obvious limitation is that file storage and access performance scalability is poor, as it is bound by the limits of the file site.

A natural evolution is then to distribute a file over multiple sites. This idea led first to the class of DFSs referred to as static partitioning schemes in Figure 16. With these schemes, some criterion is used to distribute records of a file over a number of sites. Once the distribution is established, the distribution criterion and the number of sites remain static, for the life of the file, even if updates to the file render both of these parameters sub-optimal. To change either requires redistributing the file and removing the old copy. Examples of such schemes are *round-robin* [Teradata 1988] where records of a file are evenly distributed by rotating through the nodes when records are inserted; *hashed-declustering* [Kitsuregawa, Tanaka, and Moto-Oka 1984] where records are assigned to nodes based on a hashing function; and *range-partitioning* [DeWitt et al. 1986] where key values are divided into ranges and different ranges are assigned to different nodes. With static partitioning

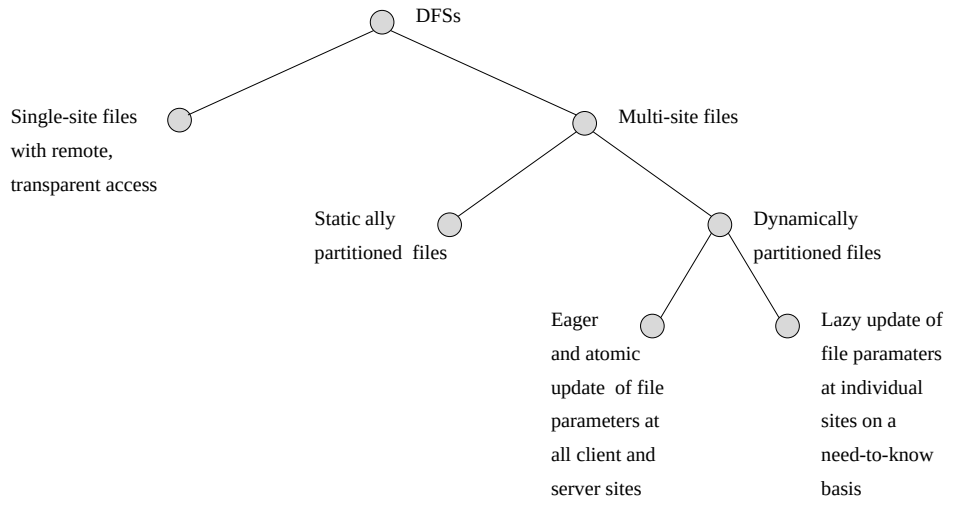


Fig. 16. Typology of Distributed File Systems.

schemes, the declustering criterion does not change over time and hence updating a directory or declustering function is not required.

To overcome the obvious limitations of the static schemes, e.g., to allow a file to expand over more sites than initially allocated, dynamic partitioning schemes started appearing. To our knowledge, the first such scheme is DLH [Severance, Pramanik, and Wolberg 1990]. This scheme was designed to take advantage of tightly-coupled multiprocessors with shared-memory. In DLH, the file is in RAM, and the file parameters are cached in the local memory of each processor. The caches are refreshed selectively when addressing errors occur, and through atomic updates to all the memories at some points during file evolution. DLH files are shown impressively efficient for high rates of insertions compared to LH.

A scheme that requires atomic updates to multiple sites, can be advantageous for performance in a tightly-coupled environment. However, it precludes a distributed file from scaling beyond a small number of sites. SDDSs were proposed to circumvent this constraint [Litwin, Neimat and Schneider 1993], LH* being the initial scheme to prove the feasibility of such an approach. Since that time, several other SDDSs based on hashing have been proposed. One such algorithm is Distributed Dynamic Hashing (DDH) [Devine 1993]. The main idea, with respect to LH*, is that DDH allows greater splitting autonomy by immediately splitting overflowing buckets. DDH uses the Dynamic Hashing (DH) of [Larson 1978] as its kernel algorithm. In DH, the dynamic hashing function is a trie. The level i basically means that the rightmost i bits of the key, or of hashed key (pseudo-key), are used as the logical address of the key. The hashing logically consists of the traversal of the trie from the root, which always points to bucket 0, to some leaf with level i, which points to the actual bucket where the key should be.

Unlike DH, though, in DDH the actual trie is not maintained. Instead, every client has an image trie that it builds through IAMs. A new client always sends a

key to bucket 0. As with LH*, a key incorrectly sent to a server is forwarded to the correct server. Bucket 0 forwards it to bucket 1 if the split using h_1 would do so, otherwise it applies h_2 and perhaps forwards it to bucket 2, etc. The process continues until the correct bucket is reached. As in LH*, every bucket keeps in its header the level i of the most recent hashing function used.

The basic versions of LH* and DDH have about the same load factor. The main advantage of DDH is that no bucket needs to store overflow records, as buckets split immediately. Hence there is no need for a split coordinator in DDH. This also leads to slightly better access performance for a client with an image reflecting the actual state of the file. This holds even with respect to LH* with presplitting, as presplits are not visible to clients. In contrast, it requires the storage for the trie on the client, which should be in practice on the order of kilobytes. Also, the key forwarding can require more messages in DDH. For a new client, it is order of $\log I$, where I denotes the maximal length of the trie, i.e., the max i. See [Devine 1993] for the corresponding simulation results that show that if new clients are unlikely, the average access performance of DDH is close to that of LH* under random searches and inserts.

The recent work of [Vingralek, Breitbart, and Weikum 1994] also defines an SDDS. It extends LH* and DDH to more efficiently control the load of a file. The goal is to limit the number of servers to the smallest possible, while still maintaining good access performance. This is accomplished by a more flexible load-balancing strategy. The *file advisor* in [Vingralek, Breitbart, and Weikum 1994] attempts to control server load as opposed to bucket load. This is enabled by having servers manage many small buckets and by allowing the advisor to *migrate* buckets from one server to another. Furthermore, the split operation applies to a server, i.e., all buckets on the server are split. This out-of-order splitting necessitates maintaining an address table (as in DDH) to map a bucket to its server and to keep track of each bucket's level. Finally, the file advisor has a more elaborate model of file load than in LH*. When it receives a collision message, it estimates the number of keys that were received by the other non-overloaded servers in order to estimate the overall load.

7. CONCLUSION

We have shown that LH* is an efficient, scalable, distributed data structure (SDDS). The analysis showed that it takes one message in the best case and three messages in the worst case to insert an object into an LH* file using the basic scheme. Correspondingly, it requires two messages to retrieve an object by its key in the best case and four in the worst case. The simulations showed that average performance is very close to optimal for both insert and retrieve requests. Hence, the performance of any algorithms that use a centralized directory has to be worse than the average performance of LH*.

LH* was also shown to have excellent performance even when a client's view of the file parameters is not up to date. LH* quickly adjusts incorrect client views of file parameters until they converge to the correct view.

We have further analyzed several variants of the basic LH* scheme. The first of these demonstrates the feasibility of LH* without any centralized control. The other variants, concurrent splits and presplits, provide techniques to increase throughput

and to more quickly adapt to data skew. The analyses have shown that all the schemes should perform efficiently for large, million-record files. Altogether, LH* appears especially efficient in its use of interconnected RAMs and should have numerous applications: very large distributed object stores, network file systems, content-addressable memories [Gallant 1992], parallel hash joins [Amin, Schneider, and Singh 1994], and, in general, for next generation databases. Operations that were impossible in practice for a centralized database may become feasible with LH*.

There are many areas of further research for LH*. The evaluation of an actual implementation is necessary. For example, we ignored the internal organization of LH* buckets. As buckets can be several megabytes large, their organization has performance implications. One attractive idea is that of buckets of different size, depending on bucket address. Similarly, the problem of handling messages at a server was not elaborated, the assumption being that messages are queued and serviced in their arrival order. For applications where different requests may have different priorities, or where contention is high, this scheme may need to be modified.

One should also investigate other SDDSs, e.g., based on other dynamic hashing schemes [Enbody and Du 1988; Salzberg 1988; Samet 1989]. SDDSs preserving a lexicographic order are also of great potential for improving the processing of range queries, as shown by the first studies in [Litwin, Neimat and Schneider 1994; Kroll and Widmayer 1994].

ACKNOWLEDGMENTS

We would like to thank Julien Levy for performing the thousands of simulations necessary for this work. We also thank the anonymous referees for their valuable comments, which helped us improve the paper.

REFERENCES

B.W. Abeysundara and A.E. Kamal. High-speed local area networks and their performance: A survey. *ACM Computing Surveys*, 23(2), June 1991.

M. B. Amin, D. A. Schneider, and V. Singh. An adaptive, load balancing parallel join algorithm. *6th International Conference on Management of Data*, Bangalore, India, December, 1994.

The fastest LAN alive. *Byte*, pages 70–74, June 1992.

R. Devine. Design and implementation of DDH: A distributed dynamic hashing algorithm. In *Proc. of the 4th Intl. Conf. on Foundations of Data Organization and Algorithms (FODO)*, 1993.

D. DeWitt and J. Gray. Parallel Database Systems: The future of high performance database systems. *Communications of the ACM*, 35(6), June 1992.

D. DeWitt, R. Gerber, G. Graefe, M. Heytens, K. Kumar, and M. Muralikrishna. GAMMA: A high performance dataflow database machine. In *Proc. of VLDB*, August 1986.

R. Enbody and H. Du. Dynamic hashing systems. *ACM Computing Surveys*, 20(2), June 1988.

R. Fagin, J. Nievergelt, N. Peppenger, and H.R. Strong. Extendible Hashing–A Fast Access Method for Dynamic Files. *ACM Transactions on Database Systems*, 4(3):315–344, 1979.

J. Gallant. FDDI routers and bridges create niche for memories. In *EDN*, April 1992.

D.E. Knuth. *The Art of Computer Programming*. Reading, MA: Addison-Wesley, 1973.

M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Architecture and performance of relational algebra machine GRACE. In *Proc. of the Intl. Conf. on Parallel Processing*, Chicago, 1984.

B. Kroll and P. Widmayer. Distributing a search tree among a growing number of processors. In *Proc. of ACM-SIGMOD*, May 1994.

P.A. Larson. Dynamic hashing. *BIT*, pages 184–201, 1978.

P.A. Larson. Linear hashing with partial expansions. In *Proc. of VLDB*, 1980.

P.A. Larson. Dynamic hash tables. *Communications of the ACM*, 31(4):446–57, April 1988.

W. Litwin. Linear hashing: A new tool for file and table addressing. In *Proc. of VLDB*, Montreal, Canada, 1980. Reprinted in *Reading in Database Systems.* M. Stonebraker ed., Morgan Kaufmann, 2nd ed., 1995.

W. Litwin, M.-A. Neimat, and D. A. Schneider. LH*—linear hashing for distributed files. In *Proc. of ACM-SIGMOD*, May 1993.

W. Litwin, M.-A. Neimat, and D. Schneider. RP*: A family of order-preserving scalable distributed data structures. In *Proc. of VLDB*, September 1994.

E. Levy and A. Silberschatz. Distributed file systems: Concepts and examples. *ACM Computing Surveys*, 22(4), December 1990.

K. Ramamohanarao and R. Sacks-Davis. Recursive linear hashing. *ACM Transactions on Database Systems*, 9(3):369–391, 1984.

B. Salzberg. *File Structures.* Prentice Hall, 1988.

H. Samet. *The design and analysis of spatial data structures.* Addison Wesley, 1989.

Herb Schwetman. Csim reference manual (revision 14). Technical report ACT-ST-252-87, Rev. 14, MCC, March 1990.

C. Severance, S. Pramanik, and P. Wolberg. Distributed linear hashing and parallel projection in main memory databases. In *Proc. of VLDB*, 1990.

M. Stonebraker. The case for shared nothing. *Database Engineering*, 9(1), 1986.

Tanenbaum. *Distributed Operating Systems.* Prentice Hall 1995.

DBC/1012 data base computer concepts and facilities. Teradata Document C02-001-05, Teradata Corporation, 1988.

D. Vaskevitch. Database in crisis and transition: A technical agenda for the year 2001. In *Proc. of ACM-SIGMOD*, May 1994.

R. Vingralek, Y. Breitbart, and G. Weikum. Distributed file organization with scalable cost/performance. In *Proc. of ACM-SIGMOD*, May 1994.

APPENDIX A. DEFINITION OF TERMS

term	description	typical value
i	file level	initially 0, increases monotonically
i'	client's view of i	0 — i
n	split pointer	0 to $2^i - 1$
n'	client's view of n	0 — n
M	total no. of buckets (servers) in file	$2^i + n$
a	a server's address	$[0, 1, 2, ..., M-1]$
j	servers' file level for local bucket	i or $i+1$
C	key value	random; 0 — 1,000,000
h_i	series of hash functions	$C \bmod N * 2^i$
α	actual load factor of file	0.6 — 1.0
α'	estimated load factor of file	
param	**description**	**typical value**
b	bucket capacity (objects per bucket)	50—10,000
t	requested load factor of file	0.7—1.0
N	initial no. of buckets in file	1