LH* — Linear Hashing for Distributed Files

Witold Litwin[*] Marie-Anne Neimat Donovan A. Schneider

Hewlett-Packard Labs
1501 Page Mill Road
Palo Alto, CA 94304
email: *lastname*@hplabs.hp.com

Abstract

LH* generalizes Linear Hashing to parallel or distributed RAM and disk files. An LH* file can be created from objects provided by any number of distributed and autonomous clients. It can grow gracefully, one bucket at a time, to virtually any number of servers. The number of messages per insertion is one in general, and three in the worst case. The number of messages per retrieval is two in general, and four in the worst case. The load factor can be about constant, 65–95%, depending on the file parameters. The file can also support parallel operations. An LH* file can be much faster than a single site disk file, and/or can hold a much larger number of objects. It can be more efficient than any file with a centralized directory, or a static parallel or distributed hash file.

1 Introduction

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 and/or from distributed processing on a number of sites.

The rationale is that whatever capabilities a single processor or site could have, a pool of sites can provide more resources. An enabling factor is the existence of high speed links. 10 Mb/sec (Megabits per second) Ethernet links are common, 100 Mb/sec FDDI or TCNS are in mass production, and 100 Mb-1Gb/sec links are coming, e.g., Ultranet and HIPPI. Similar speed cannot be achieved using magnetic or optical disks. It becomes more efficient to use a RAM of another processor than to use a local disk. Furthermore, many organizations have 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 dozens of gigabytes (GB). Such RAM files in conjunction with parallel processing should allow a DBMS to perform operations not feasible in practice within the classical database architecture.

However, distributed processing should be applied wisely. 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. We are interested in methods for gracefully adjusting the number of sites or processors involved. We propose 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 RAMs 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., 10–10,000. 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

[*]Paris 9, visiting HP-Labs and UC-Berkeley

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 a *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 net, while maximizing the load factor. We propose an SDDS called **LH***. LH* is a generalization of Linear Hashing (LH) [Lit80]. LH, and its numerous variants, e.g. [Sal88, Sam89], were designed for a single site. LH* can accommodate any number of clients and servers, and allows the file to extend to any number of sites with the following properties:

- like LH, the file can grow to practically any size, with the load factor about constant, between 65–95% depending on file parameters [Lit80],
- 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.

One variant of LH, *Distributed Linear Hashing* (DLH), is designed specifically for a tightly coupled multiprocessor site with shared memory [SPW90]. In DLH, the file is in RAM, and the file parameters are cached in the local memories of processors. The caches are refreshed selectively when addressing errors occur and through simultaneous 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. However, the need for the simultaneous updates precludes a DLH file from being an SDDS, and so does not allow it to scale beyond a small number of sites.

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.

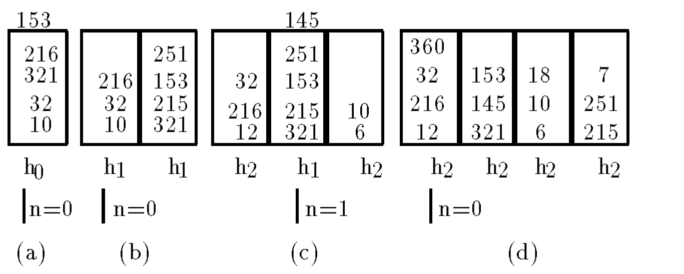
Section 2 discusses Linear Hashing and Section 3 describes LH*. Section 4 presents a simulation model of LH*. Section 5 concludes the article.

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. The files are organized into buckets (pages) on a disk [Lit80], or in RAM [Lar88]. 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: $h_i \; : \; C \rightarrow C \; mod \; N * 2^i$

Under insertions, the file gracefully expands, through the splitting of one bucket at a time into two buckets. The 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 an OID. The value of n grows one by one with the file expansion (more precisely it grows from 0 to N, then from 0 to $2N$, etc.). It indicates the next bucket to split and it is always the leftmost bucket with h_i. Figure 1 illustrates this process using h_i above, for $N = 1$ and a bucket capacity of 4.

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 shown through performance analysis is an almost constant access and memory load performance, regardless



(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.

Figure 1: Linear Hashing.

of the number of insertions. This property is unique to LH schemes. The access performance stays close to one disk access per successful search and the load factor can reach 95%.

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:

$$\begin{aligned} &a \leftarrow h_i(C); \qquad \textbf{(A1)} \\ &\text{if } \ a < n \text{ then } \ a \leftarrow h_{i+1}(C); \end{aligned}$$

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*.

An LH file can also shrink gracefully with deletions through bucket merging, which is the inverse of splitting [Lit80].

3 LH*

3.1 File expansion

We describe the basic LH* scheme, on which many variants are possible. 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. Buckets (and servers) are numbered $0, 1, ...,$ where the number is the *bucket address*. These logical addresses are mapped to server addresses as discussed in Section 3.4.

An LH* file expands as an LH file. Each key should be at the address determined by (A1). Initially, the file consists of bucket 0 only, the pointer value is $n = 0$, and h_0 applies. Thus, the addressing using (A1) uses the values $n = 0$ and $i = 0$. When bucket 0 overflows, it splits, bucket 1 is created, and h_1 is used. The addressing through (A1) now uses the values $n = 0$ and $i = 1$.

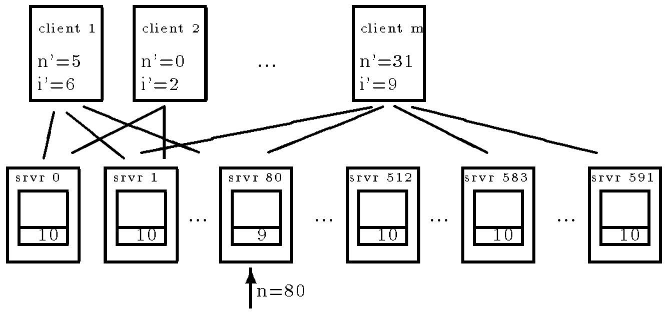


Figure 2: Principle of LH*.

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 using (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

3.2.1 Overview

Objects of a 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. 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 every client should receive a message with a new value of n after 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 542 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., one 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 a 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*. The 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. The recipient of the forwarded key checks again, and perhaps resends the key[1]. 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 *adjustment message*. 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 adjustment messages 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.

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

[1]If the file can shrink, a server may occasionally send a key to a bucket that does not exist any more. See [LNS93] for the discussion of this case.

$(A1')$ 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 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. (See [LNS93] for a discussion of bucket merging.) Thus, every key sent by a client of a 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 a 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.

Proposition 3.1 *Algorithm (A2) finds the address of every key C sent through $(A1')$, and C is forwarded at most twice.*

The following examples facilitate the perception of the proposition, and of its proof, immediately afterwards.

Examples: 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[2].

In Figures 3c-d, 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) $n \le a < 2^i$, or (ii) $a < n$ or $a \ge 2^i$. Let it be case (i), then $j = i$. It can happen that $a'' \ne a$, consider then that the forward to a'' occurs. If $a'' \ne a$, 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 $a'' \ge a$. 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 $a' > a''$, $a' \ge 2^i$, 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

[2]We assume a reliable network that does not unreasonably delay messages. Hence, we assume a forwarded key reaches bucket p before p is split using h_{i+2}. The latter split can only occur after all the buckets preceding p are also split using h_{i+2}.

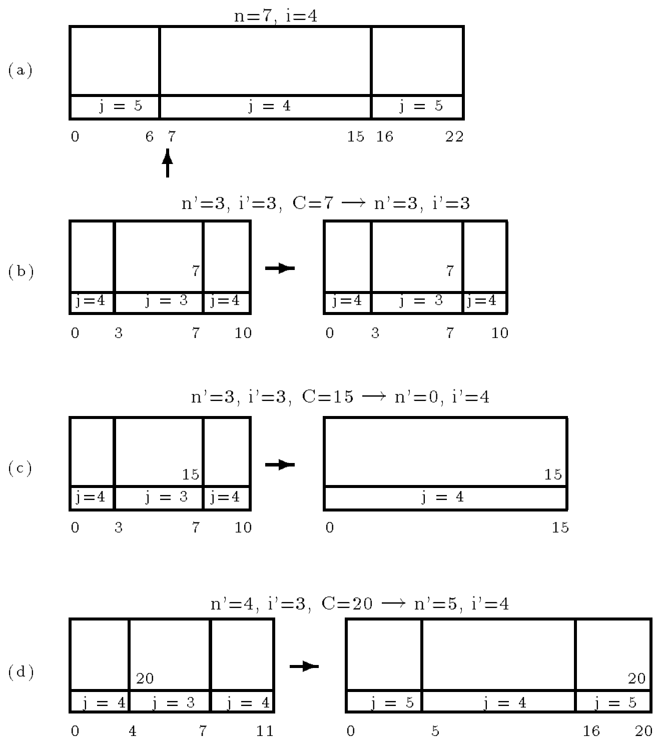


Figure 3: Images of a LH* File.

(a) – actual file

(b) – inaccurate image, but no addressing error

(c, d) – image adjustments

trigger the forward to a'', if $a'' > a$.

As the examples showed, the rationale in forwarding using $j-1$ is that the forwarding using j could send a key beyond the file. Note that several variants of (A2) can be designed. For instance, one may observe that in (A2), the calculation of a'' may be used to indirectly recognize that C was already forwarded once. If a server can know this from the incoming message, the calculation of a'' by the second recipient is useless. On the other hand, if the client sends i' along with C, then if $i' \ge j$, the calculation of a' can be eliminated.

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 the adjustment message with 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 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.

1. if $j > i'$ then $i' \leftarrow j - 1$, $n' \leftarrow a + 1$; **(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 no client inserts objects, (A3) makes every n' converge to n. Otherwise, for each client there may be a gap 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.

Examples: 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 led to an image that was accurate everywhere but at two buckets: 5 and 6. Hence the probability of an addressing error became 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 splitting can be *uncontrolled*, i.e., for each collision. Alternatively it can be *controlled*, e.g., occurring only when the load factor reaches some threshold, e.g., 80%, leading to a constant load factor in practice [Lit80]. The values of n and i can be maintained at a site that becomes the *split coordinator*, e.g., server 0. For uncontrolled splits, the split coordinator receives a message from each site that undergoes a collision. This message triggers the message "you split" to site n, and, assuming $N = 1$, it triggers the LH calculation of new values for n and i by the split coordinator using:

$$\begin{aligned}&n \leftarrow n + 1; && \textbf{(A4)}\\ &\text{if } \; n \geq 2^i \; \text{ then } \; n \leftarrow 0, \; i \leftarrow i + 1;\end{aligned}$$

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 coordinator 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 the time being, beyond the file.

Several options exist for handling splits [LNS93]. Most importantly, the split coordinator can be eliminated and splits can be done in parallel.

3.4 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. There are two approaches: (i) a static table known to all clients and servers, and (ii) a dynamic table that can become arbitrarily large, perhaps encompassing the entire INTERNET [LNS93].

3.5 Parallel operations

A parallel operation on a LH* file F is an operation to be performed on every bucket of F. Examples include a selection of objects according to some predicate, an update of such objects, a search for sets of keys in a bucket to perform a parallel hash equijoin, etc. An interesting characteristic of LH* is that a client might not know all the buckets in the file. In [LNS93], we show that the cost of a parallel operation on a file of M buckets is at most $2M + 1$ messages delivered between 1 and $O(\log_2 M)$ rounds of messages.

3.6 Performance

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 in the next 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 usually be infrequent, making the average performance close to one message. Furthermore, messages associated with bucket splitting (discussed in the following section) do not slow insertions as they are performed asynchronously.

The load factor of a LH* file is as for LH, i.e., 65–95% [Lar80, Ou91].

4 Simulation modeling of LH*

We constructed a simulation model of LH* in order to gather performance results that were not amenable to analysis in Section 3. We show that average case performance is very close to the best case for insert and retrieve operations, 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 our performance analysis. The simulator used the CSIM simulation package [Sch90].

The logical model of the simulator contains the following components. The **clients** model users of the LH* file and 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 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.

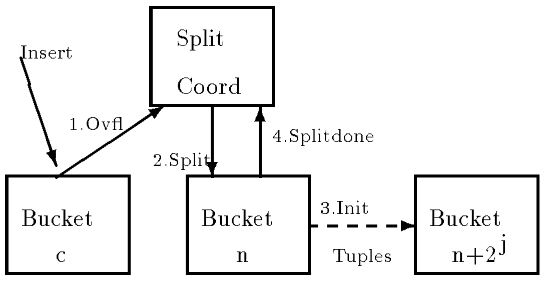


Figure 4: Splitting of a Bucket.

4.1 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 is piggybacked onto the client reply. If acknowledgments are not required, a server sends the adjustment message directly to the client. In the case of retrieves, adjustment messages are always piggybacked onto the client reply. In any case, a client uses the information in an adjustment message to update its view of the file (Algorithm A3 in Section 3.2.4).

Servers: Each bucket in a LH* file is managed by a distinct server. Servers execute algorithm (A2) to determine whether they should process the operation or forward it to a different server. If forwarding is required, an adjustment message 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 tuples in its local bucket and transfers those that rehash to the new bucket. In the current implementation, this complete operation requires a single message. When the transfer of tuples is done, a **splitdone** message is sent back to the coordinator.

Split coordinator: In our implementation, the split coordinator controls file evolution using uncontrolled splitting (see Section 3.3). The actual flow of messages required to split a bucket is shown in Figure 4. As is shown, four messages are required for each bucket

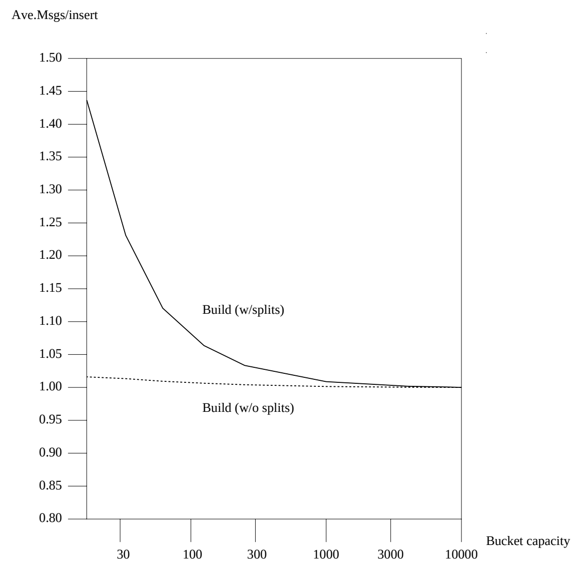


Figure 5: Performance of file creation.

split. This is a slight simplification because it could take several messages to transfer the keys to a new bucket, when bucket capacities are large. Furthermore, the coordinator only allows a single bucket to be undergoing a split operation. All collision notification messages received from servers while a split is in progress are queued for later processing.

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.

4.2 Experimental results

4.2.1 Performance of file creation and search

Figure 5 shows the average cost, in messages per insert, to build LH* files with bucket capacities ranging from 10 to 10,000 keys. Each file was constructed by inserting 10,000 random keys, thus resulting in files with the number of buckets ranging from over 1,000 down to 1. Inserts did not require acknowledgements. The lower curve plots the average number of messages per inserted key as seen by the client, i.e., it consists of the original message from the client to the server, forwarding messages from server to server, and finally, adjustment messages from servers back to the client. This curve confirms our performance predictions in Section 3. First, performance is better for files with larger bucket capacities, although in this case performance is quite stable. And second, insert performance is very close to the best possible — one message per insertion. The figure shows that the difference is under 3%.

The upper curve shows the complete load on the networking system for building LH* files. That is, it includes the messages from the lower curve plus the messages associated with bucket splitting (four messages per split). Since inserts can take place concurrently with bucket splits, this metric should not adversely affect client performance (as shown by the lower curve) unless the network becomes a bottleneck. Note that the curve converges rapidly towards the lower one anyway.

Table 1 presents a more detailed picture of the curves in Figure 5. In addition to the average messages per key insert (**AvMsgs**), which forms the upper curve in Figure 5, it includes the number of addressing errors incurred for the key inserts during file creation (column **Errs**). As is shown, the number of addressing errors is very small, even when the bucket capacity is small.

The column **Msgs-ack** in Table 1 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 strong guarantees on the success of their updates. As is shown, these numbers are almost exactly one greater than the previous case where inserts are not acknowledged. The reason for being slightly less than one is due to the piggybacking of addressing error messages onto the acknowledgement messages to the client.

Finally, column **Search** shows the average performance of a client retrieving random keys from the files. For each bucket capacity the client first inserted 10,000 random keys. It then reset its view of the file to empty and retrieved 1,000 keys. This was repeated 100 times and the results were averaged. As the table shows, it generally 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%.

4.2.2 Convergence of a client view

In this set of experiments we were interested in determining how fast a read-only client, starting with a view of the file as empty, obtains a true view of the file (using the image adjustment algorithm (A3)). Two metrics are of interest: the number of addressing errors incurred before converging to the true state of the file, and the number

Bkt Cap	No. of Bkts	Build			Search
		Errs	AvMsgs	Msgs-ack	AvMsgs
17	1012	161	1.437	2.421	2.008
33	512	134	1.231	2.218	2.007
62	255	94	1.120	2.111	2.007
125	128	64	1.064	2.057	2.006
250	64	41	1.033	2.029	2.006
1000	16	14	1.009	2.007	2.004
4000	4	3	1.002	2.002	2.002
8000	2	1	1.001	2.001	2.001

Table 1: File build and search performance (10K inserts, 1K retrieves).

Bkt Cap	No. of Bkts	Addr Errors		Retrieves	
		Ave	Std	Ave	Std
25	7296	9.3	2.6	3995.8	2813.5
250	512	6.8	2.4	476.5	464.1
2500	64	5.1	1.6	69.3	60.5

Table 2: Convergence of a client view (100K inserts).

of objects retrieved before convergence is reached.

Table 2 presents the detailed results for files with bucket capacities ranging from 25 to 2500 (each file was populated with 100,000 random keys). The client was run 100 times, each time starting with an empty view of the file, and the results were averaged. The results, under the column **Addr Errors**, show that it takes relatively few addressing errors before the client's image converges to the true state, even when the capacity of each bucket is small and hence the number of buckets is large. In fact, the average number of addressing errors is slightly less than $\log_2$ of the number of buckets. This is intuitive, because, on average, each addressing error halves the number of buckets that the client may address incorrectly. Note also that performance is better for files with large bucket capacities.

Furthermore, the results under the column **Retrieves** demonstrate that a client can retrieve many objects without incurring addressing errors even though the client's view of the file is inaccurate. For example, when the bucket capacity was 25, the client retrieved 3,996 objects before its view of the file matched the true state of the file. Since only 9 addressing errors were incurred in order to reach convergence in this case, 3987 of the 3996 objects were retrieved without error, for an average of just 2.002 messages per retrieve.

Insert Ratio	Client 0 (Active)			Client 1 (Less Active)		
	AvMsg	Errs	%Errs	AvMsg	Errs	%Errs
1:1	2.01	125	1.25%	2.01	126	1.26%
10:1	2.01	115	1.15%	2.05	49	4.90%
100:1	2.01	104	1.04%	2.23	23	23.00%
1000:1	2.01	104	1.04%	2.50	4	40.00%

Table 3: Two clients (bucket capacity = 50).

Insert Ratio	Client 0 (Active)			Client 1 (Less Active)		
	AvMsg	Errs	%Errs	AvMsg	Errs	%Errs
1:1	2.004	38	0.38%	2.004	39	0.39%
10:1	2.002	20	0.20%	2.013	13	1.30%
100:1	2.002	20	0.20%	2.100	10	10.00%
1000:1	2.002	20	0.20%	2.500	5	50.00%

Table 4: Two clients (bucket capacity = 500).

4.2.3 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 experiences 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 first client inserts N keys for every key inserted by the second client (N is referred to as the *Insert Ratio*). The first client always inserts 10,000 keys. Thus, with an insert ratio of 100 to 1, the second client only inserts 100 keys. All insert operations required an acknowledgment.

The results are summarized in Table 3 for LH* files with a capacity of 50 keys at each bucket. The average number of messages per insert, the total number of addressing errors, and the percentage of addressing errors related to the number of inserts are shown for each client. As the table shows, the performance of the second client degrades as it is made progressively less active. This occurs because the inserts by the first client expand the file thus causing the second client's view of the file to be outdated. This then results in the second, less active client experiencing an increased percentage of addressing errors.

Table 4 repeats the experiments of Table 3 with the exception that the bucket capacity has been increased from 50 to 500. The overall result from this experiment is that the percentage of addressing errors decreases with larger bucket capacities because greater capacities result in files with fewer buckets and hence it is less likely that a slower client will have an outdated view of the file.

(For example, with an insert ratio of 1:1, the number of buckets was decreased from 579 to 64 when the bucket capacity was increased from 50 to 500.) However, a comparison of the two tables shows that the percentage of addressing errors does not increase significantly for files with smaller bucket capacities.

An experiment with 100,000 keys and with settings identical to that of Table 4 showed that performance of less active clients tends to be better for large files [LNS93]. For example, the percentage of errors observed by the less active client was reduced to 17% and the message cost was decreased to 2.17, for an insert ratio of 1000:1.

4.2.4 Marginal costs during file growth

Many file access methods incur high costs at some points during file evolution. For example, in extendible hashing, an insertion triggering the doubling of the directory incurs a much higher cost. In LH*, the cost of file evolution is rather stable over the lifetime of a file. Experiments computing marginal costs reported in [LNS93] show that access performance stays between 2.00 and 2.06 messages over the lifetime of the file, even for files with small bucket capacities.

5 Conclusion

LH* is an efficient, extensible, distributed data structure. An LH* file can grow to virtually any size. In particular, the algorithm allows for more efficient use of interconnected RAMs and should have numerous applications: very large distributed object stores, network file systems, content-addressable memories, parallel hash joins, and, in general, for next generation databases. Operations that were not possible in practice for a centralized database may become feasible with LH*.

Our analysis showed that it takes one message in the best case and three messages in the worst case to insert a key into a LH* file. Correspondingly, it requires two messages to retrieve a key in the best case and four in the worst case. Furthermore, through simulations we showed that average performance is very close to optimal for both insert and retrieve queries. Hence, performance of any algorithms that use a centralized directory has to be worse than the average performance of LH*.

There are many areas of further research for LH*. Variants of the basic LH* scheme outlined in this paper and in [LNS93] should be analyzed in greater depth. Applications of LH*, e.g., hash-joins and projection, should be examined. Concurrent use of LH*, e.g., on the basis of [Ell87], and fault tolerance are especially interesting areas. The evaluation of an actual implementation would also be interesting. For example, we ignored the internal organization of LH* buckets. As buckets can be several megabytes large, their organization could have many performance implications. One attractive idea is that of buckets of different size, depending on bucket address.

Finally, one should investigate other SDDSs, e.g., based on other dynamic hashing schemes, [ED88, Sal88, Sam89], or preserving a lexicographic order, e.g., B-trees or [Hac89, Kri86, LRLH91], that can improve the processing of range queries.

Acknowledgements

We would like to thank Spyros Potamianos for his invaluable help in formatting the paper.

References

[ED88] R. Enbody and H. Du. Dynamic hashing systems. *ACM Computing Surveys*, 20(2), June 1988.

[Ell87] Carla S. Ellis. Concurrency in linear hashing. *ACM TODS*, 12(2), June 1987.

[Hac89] N.I. Hachem, et.al. Key-sequential access methods for very large files derived from linear hashing. In *Intl. Conf. on Data Engineering*, 1989.

[Kri86] H.-P. Kriegel, et.al. Multidimensional order preserving linear hashing with partial expansions. In *Intl. Conf. on Database Theory*. Springer-Verlag, 1986.

[Lar80] P.A. Larson. Linear hashing with partial expansions. In *Proc. of VLDB*, 1980.

[Lar88] P.A. Larson. Dynamic hash tables. *CACM*, 31(4), April 1988.

[Lit80] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Proc. of VLDB*, 1980.

[LNS93] W. Litwin, M.-A. Neimat, and D. Schneider. LH*—linear hashing for distributed files. Tech. report HPL-93-21, Hewlett-Packard Labs, 1993.

[LRLH91] W. Litwin, N. Roussopoulos, G. Levy, and W. Hong. Trie hashing with controlled load. *IEEE Trans. on Software Engineering*, 17(7), 1991.

[Ou91] S.F. Ou, et.al. High storage utilisation for single-probe retrieval linear hashing. *Computer Journal*, 34(5), Oct. 1991.

[Sal88] B. Salzberg. *File Structures.* Prentice Hall, 1988.

[Sam89] H. Samet. *The design and analysis of spatial data structures.* Addison Wesley, 1989.

[Sch90] H. Schwetman. Csim reference manual (revision 14). Tech. report ACT-ST-252-87, Rev. 14, MCC, March 1990.

[SPW90] C. Severance, S. Pramanik, and P. Wolberg. Distributed linear hashing and parallel projection in main memory databases. In *Proc. of VLDB*, 1990.