LH*$_S$: a High-availability and High-security Scalable Distributed Data Structure

W. Litwin[1], M-A Neimat[2], G. Levy[3], S. Ndiaye[3], T. Seck[3]

Abstract

LH$_S$ is high-availability variant of LH*, a Scalable Distributed Data Structure. An LH*$_S$ record is striped onto different server nodes. A parity segment allows to reconstruct the record if a segment fails. The insert or key search time is about a msec on a 10 Mb/s net, and about 100 μs at 1 Gb/s net, assuming the segments in the distributed RAM. The file size depends only on the distributed storage available, i.e., a RAM file can reach dozens of GB in practice. Data security is enhanced, as every site contains only partial and typically meaningless data. The price to pay is 20 - 50 % more storage for the file than for an LH* file, and some additional messaging, especially for the scan search.*

Introduction

Multicomputers are collections of autonomous WSs or PCs over a network (*network multicomputers*), or of share-nothing processors with a local storage linked through a high-speed network or bus (*switched multicomputers*) [T95]. It is well known that multicomputers offer best price-performance ratio [T95], [M96]. Research on multicomputers becomes popular [C94], [G96]. The *Scalable Distributed Data Structures* (SDDSs), like LH* [LNS93], are new data structures designed for multicomputer files. An SDDS gracefully scales up with inserts over available distributed storage, the distributed RAM storage preferably. One problem that a designer of an SDDS may face is a site failure. Some applications require high-availability schemes, allowing data to remain available despite a site failure [M96]. Distributed data are also vulnerable to an unauthorized local or remote intrusion. This makes useful the high-security SDDS schemes, making an unauthorized access to the data difficult.

The LH* schemes with mirroring in [LNS96], called here LH*$_M$, are first SDDSs designed for high-availability. The schema proposed below, termed LH*$_S$, responds to the high-availability and the high-security needs. A record in LH*$_S$ file is striped into $k > 1$ segments (stripes) put on different nodes, and into distinct LH* files. There is also a segment with the parity bits, as in RAID schemes and others [PGK88], [HO95], [R94], [SS90]. The striping is basically performed at bit level, putting consecutive bits of the record into different segments. The schema supports any single bucket (server site) unavailability. It also supports any single-site intrusion without disclosing a record content. One can read at best one segment, typically meaningless, as containing 1 bit from every k in the record.

With respect to an LH* file, the LH*$_S$ file with the same records requires more storage, usually about $15 \div 25$ %, because of the parity segments. Access performance of LH*$_S$, in terms of network transfer time per insert or key search, is close to this of LH*. There is some deterioration for an insert, as the parity segment has to be sent out. Similarly the key search for a record can be somehow slower than for LH*, as it has to be sent out by the client to k buckets. There is also more CPU time involved as any record travels in at least two messages. Nevertheless, this price should be acceptable for many applications.

The bit-level striping affects more the scan search, where all the records are searched for some non-key values. A scan search in an LH* file is dealt with using a parallel query to every bucket. It requires in general a more costly processing in an LH*$_S$ file with the bit-level striping, as records have to be reconstructed on-the-fly. For applications where scan performance is of prime importance, LH*$_S$ allows for striping at the attribute-level. A segment contains then entire attributes of the record. Scan search performance becomes better, at the expense of the high-security, as an intruder to a site disposes at least of some attributes of a record.

Next section presents LH*$_S$. Section 3 discusses the performance of file manipulations. Section 4 discusses the security issues. Section 5 overviews the related work. Section 6 concludes the paper.

[1] Université Paris 9, litwin@etud.dauphine.fr

[2] Hewlet-Packard Laboratories, Palo Alto, California, neimat@hpl.hp.com

[3] Université de Dakar, ndiayesa@esp.esp.sn, seckm@ensut.ensut.sn

Overview of LH*S

*Principles of LH**

We now recall the principles of LH* schemes [LNS93]. An LH* file resides on *server* computers (nodes), and is accessed by applications on *client* nodes. A server is always available for access from the clients. A client in contrast is autonomous, perhaps mobile, hence guaranteed to be accessible only when it is an initiator of the connection. The file consists of records identified by (primary) keys. Records are stored in *buckets* with a *capacity* of b records ; $b >> 1$. Buckets are numbered 0,1,2..N. There is one bucket of a file per server, although different files may share servers. Buckets are assumed in RAM. The file starts with bucket 0, and scales up with inserts, through bucket splits.

Bucket addresses are mapped to the network addresses of the servers using *physical allocation tables* at the clients, and the servers. Each element of a table contains an address. A table, let it be T, can be *static* or *dynamic*. In the latter case, the address for bucket n can be arbitrarily chosen, especially by the coordinator, and stored in $T(n)$. Different sites may have different tables. The coordinator refreshes T at every bucket, when it sends the request to split. The message contains then all the new addresses added to the file since the previous split of the bucket. The servers send T to clients with every IAM. A dynamic table can scale potentially to any length. Also, it allows for easier bucket migrations than if a static T.

The splitting and addressing rules of LH* are based on those of *linear hashing* (LH) [L80]. Every split moves about half of the records in a bucket into a new bucket at a new server, appended to the file. The splits are done in the order 0; 0, 1; 0, 1, 2; 0, ..., 2^i; 0, ... The next bucket to split is denoted bucket n, and is also called the *split pointer*.

The splits are triggered by bucket overflows. In LH*, a bucket that overflows reports the overflow to a dedicated node called the *coordinator*. The coordinator applies the *load control policy* to find whether the overflow should trigger the split. If so, the coordinator initiates the split of bucket n.

To perform the splits and the addressing, an LH* file uses a family of hash functions h_i, $i = 0,1..$ Each h_i hashes a key c into bucket address $h_i(c) = c \bmod 2^i$. A split results from the replacement of function h_i currently used for bucket n with function h_{i+1}. The i value is called the *bucket level*. At any time, an LH* file can only have buckets with level i or $i + 1$, $i = 0,1,..$ The coordinator is the only node in the file that knows the current values of n and i. The *correct address*, denoted a, of key c in an LH* file is the address where c should be, given n and i, i.e., where it should be *dynamically hashed*. The address a is defined by the LH addressing algorithm [L80]:

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

To avoid a hot spot, LH* clients do not access the coordinator for the address computation. As for any SDDS, an LH* client has therefore its own *image* of the file. For LH*, it consists of values noted i' and n' ; $i' = n' = 0$ for a new client. These values may vary among clients and may differ from the actual n and i. The client uses its image to calculate the address $a' = A(n', i')$ for key c, while issuing a (point-to-point) request for the search of c, or for an insert or a delete of the record identified by c. It then sends the request, and perhaps the record to server a'. LH* supports also multicast and broadcast queries addressing through one message all N buckets, [LNS93].

It might happen that $a' \neq a$. Hence, every server s receiving a request first tests whether $s = a$. For this purpose, every server keeps the current value of i. It can be proven that $s = a$ iff $s = h_i(c)$. If the test fails, the server forwards the request to another server. The LH* *test and forwarding algorithm* is as follows, [LNS93] :

(A2) $a' \leftarrow h_i(c)$;
if $a' = a$ then accept c ;
$a'' \leftarrow h_{i-1}(c)$;
if $a'' > a$ and $a'' < a'$ then $a' \leftarrow a''$;
forward c to bucket a' ;

The forwarding process could a priori create many hops. The major property of LH* is however that every request to an LH* file is delivered to the correct address after at most two hops, [LNS93].

As for any SDDS, the correct server finally sends a message back to the client, called an Image Adjustment Message (IAM). For LH*, an IAM contains the i value of server a'. The value of split pointer n is unknown to the servers, hence is not in IAMs. The client executes then the *IA-Algorithm*, [LNS93] :

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

The result of (A3) is a better image, with both i' and n' closer to the actual values. Also, as long as there is no new split, the same addressing error cannot occur. (A3) makes LH*-images converge rapidly [LNS93]. Usually, $O(\log N)$ IAMs to a new client (the worst case for image accuracy) suffice to about eliminate the forwarding. If a client already has a good image, but the file starts to scale-up, algorithm (A3) suffices to keep the incidence of forwarding on the access

performance about negligible. In practice, the average key insert cost is one message, and both a successful and unsuccessful key search cost is two messages, regardless of the file size. The worst access performance of an insert or search corresponds to the case of two hops. These costs for LH* are of four messages, also regardless of the number of nodes of the file.

The principles of LH* led to many variants [LNS93a], [KLR96]. The schemes offer various trade-offs adapted to particularities of applications.

Principles of LH$*_S$

We now discuss the basic LH*$_S$ using the *bit-level striping* (segmentation, scattering..). A *record R* is a key, usually denoted *c*, and a sequence of bits *B*, numbered from left to right
$B = b_1,...,b_k b_{k+1}...b_{2k} b_{2k+1}...b_{mk}$. The size of *B* is *mk*, last bits being padded if needed in practice. When an LH*$_S$ client should store *R*, then it proceeds as follows, Fig. 1:

- It produces *k segments*, $k > 1$. The *i*-th segment s_i consists from *c* and from all the bits s'_i :

$$s'_i = b_i\, b_{k+i}\, b_{2k+i}\, ...$$

- It produces the *parity segment* s_{k+1} that also contains *c* and the parity bits s'_{k+1}, let us say for the *even* parity:

$$s'_{k+1} = b'_1\, b'_2 .. b'_m$$

where bit b'_j is the parity bit for the string with the *j*-th bit of each segment ; $1 \le j \le k$. If some segment *s* of *R* cannot be read, the parity segment, allows to reconstruct *s*.

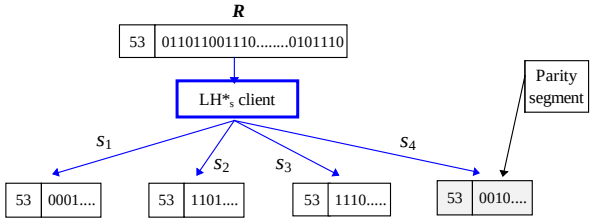

Fig. 1 LH*s scattering of a record into k = 3 segments

An LH*$_S$ file is created as a family Φ of k+1 LH* *segment files* $S_1..S_{k+1}$. File S_i stores all the segments s_i. The address of segment s_i is calculated from its key *c* that is, we recall, also the key of *R*. As in LH*$_M$, [LNS96], two segment files can be structurally-alike (SA). They have then the same parameters : the bucket size, the functions h_i, etc. They can also be structurally-different (SD) which means that these parameters differ. SD-files are *loosely-coupled* if they share functions h_i. Otherwise, they are *minimally-coupled* [LNS96]. Fig. 3 shows the relationship between SA and SD files.

The basic constraint on Φ is that for every record *R*, all its segments are mapped to different nodes, or at least buckets. One way to achieve it is to provide each *S* with the physical allocation table T_S spanning over distinct nodes of the multicomputer. In other words, no node carries then a bucket of S_i and of S_j when $i \neq j$. For SA segment files, every segment of the same record *R* is usually in the bucket with the same bucket address *m* within its segment file, as in Fig. 3. For instance, after some inserts into *F*, the segments of *R* with key $c = 100$, may be all in buckets 57 of their files. The client keeps a single LH* image with the (guessed) file level i' and the split pointer value n' for every *S*. For SD segment files, the segments' bucket addresses typically differ, Fig. 3. Hence, there is one image per *S* on the client and these images usually differ as well.

As usually for an LH* file, every *S* expands through splits, tolerates addressing errors, and sends IAMs to its clients. Splits among segment files are not synchronized, i.e., each *S* split autonomously. Hence, even in SA segment files, it may happen that bucket *m* in a segment file S_i splits before another bucket *m* in the segment file S_j ; $j \neq i$. One reason may be that S_j failed when it should split after a new insert. A segment of record *R* may then be in new bucket in S_i while another segment of *R* is still in bucket *m* in S_j. Hence, the addresses of the segments of a record within their SA segment files may sometimes differ as well.

The whole set Φ of LH*$_S$ segment files *S* has a common component at some server called the *segment file coordinator*, SC in short. Its address is known to every server and every client. SC takes care of the LH* coordination for each *S*. This includes all the allocation tables, assumed dynamic, since easier to manage for a spare production. In addition, it has capabilities for the fault-tolerance of the whole collection that we'll introduce.

In particular, SC gets alerted when a bucket failure is detected. The alert may come from a client that failed with a file manipulation. It may also come from a server that could not forward a message or could not split. If a failure is confirmed, SC coordinates the creation of the spare.

File manipulation

Inserts

To insert record *R*, the client first produces the $(k + 1)$ segments. Then. it sends each segment s_i ; $i = 1..k + 1$; using a unicast message to bucket m_i, where m_i results from the LH* address computation (A1) executed on the client for each segment file S_i. Unlike

for LH*, each message carries the value m_i for the reasons discussed more in depth in Section 0. The server addressed by the message usually carries bucket m_i. It might rarely happen that it carries another bucket. This occurs when bucket m_i failed and was recreated at another location. If it happens, the server that got the message forwards it to SC. SC determines from the allocation tables where bucket m_i actually is, and resents the message. An IAM comes later to the client, from bucket m_i, with its actual address.

Once bucket m_i gets the message, a forwarding may occur as usually for an LH* file. The forwarded message also carries the number, let it be m, of bucket m the message is intended for. If another bucket is found at the destination site instead, the message is resent to SC, as above, etc. Since in LH* there are at most two forwardings, SC can get the messages at most three time as well. Same process may occur for each segment file. It is however very unlikely that all this happens simultaneously for all the segments of the same insert. The typical case is that every segment is inserted without any forwarding.

Assuming nevertheless that a forwarding occurs at a segment file, the client receives an IAM. Up to $(k + 1)$ IAMs may therefore be triggered by an insert. The client of SD segment files adjusts each image. The client of SA segment files, has to proceed differently, since it has only one image. The basic strategy is that the client performs the IA-algorithm for an address a' only when all the $(k + 1)$ IAM messages with a' and (same) i' are received.

A client or a forwarding server may also find a bucket unavailable. It then alerts SC and forwards the segment to it. The client considers the insert successful if it encounters at most one failed bucket. Otherwise, the client waits for a message from SC, advising whether the insert is finally successful or not. The failed bucket can be indeed the correct bucket for the segment, or the intermediate bucket that should forward the segment. The insert basically fails if the SC finds unavailable more than one correct bucket for a segment. There can be in contrast several forwarding buckets unavailable. SC may bypass such buckets, as it has the actual image of the file. If only one bucket is unavailable, the segment that was passed to SC is finally inserted during the recovery procedure discussed below.

LH* supports also *bulk inserts*. A message with several records is then multicast to all the servers. Each server stores then the records whose keys correspond. LH*$_S$ file also supports the bulk inserts. A record can be sent entirely in a bulk message. Alternatively, one may spread its segments into several bulk messages. This strategy enhances the transfer security. Note that one should send also then the parity segment, computed by the client.

Splits

Splits of segment files are basically performed as for LH*. Especially, if the new bucket fails during the split, i.e., before the split is committed, the split restarts with a new target bucket. The new case is that a bucket can fail failure during the split. The split is stopped. The spare is created as for any failed bucket, using the other segment files to reconstruct all the unavailable segments, as described below. Then, the split is restarted from the beginning. Alternatively, the new bucket sends to the spare all its keys. The spare moves only the segments that should move and whose keys are not among the received ones.

Deletes

To (physically) delete R (c), the client sends the key to all the corresponding $(k + 1)$ buckets. Every bucket deletes the corresponding segment, as discussed in [LN95]. Physical deletion being rare in practice, we do not discuss them more in what follows.

Search

Key search

The search for record R, given its key c, is performed basically through sending c to k servers, $S_1, S_2 ... S_k$. The client uses k unicast messages to the buckets whose addresses result from the LH* address computation for each segment file. For the SA segment files, the bucket address is computed once for every segment file. For SD segments, there are k calculus and the results may differ. If all the segments come, the clients synthesize the record.

As for the inserts, each message carries its intended bucket number. The servers may forward the messages, as usually for LH*, or to SC, if the actual bucket does not match the intended one. If a reply is missing, despite perhaps several attempts to get it by the client, the client alerts SC. If only one reply is missing, the client issues a message to S_{k+1}. If this segment comes, the client synthesize the record. Otherwise, the search fails.

The search can be unsuccessful. In this case, it is not necessary to have all the servers to reply. The buckets perform then the hashing $m = c \bmod k$. Only the bucket within S_m replies. An alternative strategy is that no bucket replies and the client declares the search unsuccessful by time-out. It is highly unlikely that if the search was successful, the replies from all k buckets were lost on the way, and the client incorrectly understood that the search was unsuccessful.

Scan search

LH* supports also the *scan search*, or the *scan* in short, where all records are searched for some non-key values. For records that are collections of attributes, the scan search criteria usually consist of a selection predicate on the attribute values. An LH* scan search is realized through a parallel search over each bucket using the selection expression got from the client. The results are returned to the client also in parallel. A scan may terminate using a *probabilistic* (time-out) *termination* where only the buckets that have sent records reply. One may alternatively request the *deterministic termination*, where any bucket replies, with its address, and selected records or a null message if the search is unsuccessful. The client may compute whether all the buckets currently existing in the file replied. If only a few records are to be selected and the file is large, then the time-out termination is much faster than the deterministic one.

A scan is sent by the client using either unicast messages or a multicast message. In the former case, the client may not know all the corresponding addresses. An algorithm propagating then the search to all the servers of the LH* file is defined in [LNS93a]. The drawback of a multicast message is that it is received by all the sites on the net. Hence it disturbs also those not serving the file.

A scan in an LH*$_S$ file is typically a more complex operation than in an LH* file with the same records. Each bucket contains indeed only some non-consecutive bits of each record. Such a content provides the high-security, but is about meaningless for evaluating selection predicates. The only practical way to proceed is to reconstruct all the records at some servers, where the parallel scan is performed as for an LH* file. The reconstruction is essentially a multiway equi-join on the key value between all the segments. The servers where it is performed are called *join servers*. It should be worth using all the available segment file servers as join servers and uniformly. This rationale leads to the following algorithm.

(A4) LH*$_S$ scan search

1. Using unicast or multicast, the client sends the scan search Q in parallel to every bucket m in its image(s) of each segment file S_i ; $i = 1,2..k$.
2. If unicast is used, then each server applies the LH* parallel search propagation algorithm.
3. For every segment s_i (c) with key c at every server, perform the hashing $h(c) = c \bmod k + 1$. Consider the server of segment $s_{h(c)}$ (c) as the join server of the record R (c). If $i = h(c)$, then prepare for the reception of other segments of R. Otherwise, send s_i (c) to the corresponding address i n file $S_{h(c)}$.
4. For every server, perform the join of all the segments received with the corresponding segments stored locally, to reconstruct every R.
5. If any expected segment is missing, alert SC and search S_{k+1}. Reconstruct remaining R's.
6. For each server, perform Q and send the result to the client.

Details of (A4) are discussed in [LN95]. The basic way for sending the results of Q back to the client is to simply to send all the selected records by every join server. Another possibility is to send from the server of a selected record the messages to the buckets with the corresponding segments requesting them to be sent to the client. Finally, the join server may send to the client only the keys. The client searches then the corresponding segments itself. The latter approaches are more costly in CPU and messages, but offer higher security.

The client may wish the parallel search to terminate in a deterministic or probabilistic way, i.e., by time-out. The deterministic termination is costly, as every bucket has to send a result, perhaps null. For a probabilistic termination it suffices that one sends only the selected records or segments. There is no guarantee that the client gets all the records that it should.

Failure management

Overview

We consider the following kinds of failures :
1. A search at an available server does not find a segment s_i ; $i \leq k$; while the client gets all other segments.
2. A bucket is unavailable for access.

In case (1), s_i is simply reconstructed by the client on the fly, after additional search of s_{k+1}. Then, s_i is reinserted, by the client or SC. If more segments are missing, the record and the segments cannot be reconstructed from s_{k+1}. The corresponding recovery is considered application dependent and beyond the scope of the basic LH*$_S$.

In case (2), the lack of access is due to the bucket failure, or to the network failure. Once the access is reestablished, the bucket restarts the service by contacting SC. The clients are served again by the bucket only if SC informs the server that no spare was produced in the meantime.

If a bucket fails, two strategies exist :
1. One continues, using the available segments, until the bucket recovers by itself. If $k >> 1$, the difference in the workload on these segments should be negligible. If updates occur, the corresponding segments are reconstructed when the bucket is up again.
2. A spare is produced, replacing instantly the failed bucket.

The choice between strategies (1) and (2) is application dependent. Strategy (2) clearly offers higher availability. This strategy is the basic one for LH*$_S$. The spare production algorithms depend on the structure, SA or SD, of the segment files. They are based on those for the LH* with mirrors [LNS96].

Creating a spare

Let it be bucket n_1 that failed in its segment file, let it be S_1. To create the spare, one should find the segment buckets of $S_2..S_{k+1}$ that contain the remaining segments of every record R whose segment s_1 was in the lost bucket. For each lost segment, the remaining segments of R have to be joined at some join server. The lost segment may then be recomputed at the join server, and inserted to the spare being created.

To determine the addresses of the buckets with the remaining segments, we consider at first the SA or SD segment files, except for the minimally coupled files. Let j_1 be the level of bucket n_1. Every S_i uses the same hash functions. Hence, for each S_i ; $i > 1$; two cases may happen:

1. The segments are all in one bucket. This will happen if there is not yet bucket n_2 in S_i such that $n_2 = n_1$ or level j_2 of bucket n_2 is $j_2 \leq j_1$. In particular, if $j_2 < j_1$, then bucket n_2 may contain more records than bucket n_1.
2. The records are in several buckets. This will happen if $j_2 > j_1$.

In case (1), n_2 is at the largest address n_2 such that:

$n_2 = n_1$ or $n_2 = n_{2,1} = n_1 - 2^{j_1-1}$ or $n_2 = n_{2,1} - 2^{j_1-2}$ or …

… $n_2 = 0$

In case (2), buckets to be read are bucket n_1 itself, its children, children of children, etc. Hence these buckets are :

$m = n_1$ or $m = n_{1,1} = n_1 + 2^{j_1-1}$ or $m = n_{1,2} = n_1 + 2^{j_1}$ … or

$m = n_1 + 2^{j_2-1}$ or $m = n_{1,1,1} = n_{1,1} + 2^{j_1}$ or $m = n_{1,1,2} =$

$= n_{1,1} + 2^{j_1+1}$ or $m = n_{1,1,1,1} = n_{1,1,1} + 2^{j_1+1}$ ….

The first line corresponds to the addresses of the children of n_1. The next line corresponds to the children of the first child of n_1 (in some cases this line may if fact be a copy of the first line since one may have $n_1 = n_{1,1}$). Then, there are the children of the next child, etc. Fig. 2 illustrates the formulae.

Let b_i denote the bucket capacity of S_i. Let us assume that the load control policy is the same for both files. Then, typically, case (1) occurs if $b_i > b_1$ and case (2) occurs if $b_1 > b_i$. If $b_2 = b_1$, one has the case of SA segment files.

The following algorithm is executed by SC and the buckets' servers, to produce a spare for SA and SD files.

(A5) Spare creation for an LH*$_S$ file

Consider that the lost bucket is bucket n in file S_1.

1. For every segment file S_i ; $i = 2..k + 1$; SC determines as above the addresses m_i of buckets that could have the remaining segments of the record R whose segment s_1 was in bucket n. Let l be the total number of these buckets.
2. SC allocates a server for the spare and an empty new bucket n is created. This server receives from SC all the bucket addresses computed in Step 1.
3. For every m_i, SC sends the query with level j of bucket n. It requests every key c such that segment s_1 of R (c) was in bucket n, i.e., it requests to test for every c whether $n = h_j$ (c).
4. Every bucket server computes for each c found through Step (4) the hashing function h (c) = 2 + c mod l. The server of bucket m with the segment $s_{h(c)}$ is assumed the join server for the record R (c).
5. For every segment s (c) from step (4), if bucket m is not at the server computing the function, then the server sends segment s (c) to bucket m. To allow for that, the query of step (4) contains also the actual addressing parameters (i, n) of each segment file, and the physical addresses of all the buckets found through step (2).
6. For every bucket m, there is a terminating message with the number segments sent, expedited by every bucket from which bucket m expected or got segments.
7. Every record corresponding to the lost bucket is reconstructed on the join server. The lost segment is computed and sent to the spare bucket.
8. The spare server expects a termination message from each join server, containing the number of records that it should receive.
9. SC sends the physical address of the spare to the server with the parent of the lost bucket. The server updates its allocation table.
10. Through Step 4, the algorithm distributes the computation of the lost segments. A naive approach would be to centralize this computation on the spare, making it much less efficient. The goal of the terminating messages in Steps 6, 9, is the deterministic termination. Details of Step 5 - 9 are implementation depended. For slower nets, it may be advantageous send segments to the join server and from a join server to the spare in bulks. The parent allocation table is the only one that points to the spare, as the parent might need to perform a forwarding, except for that of SC. Hence, it is the only bucket sever table that needs an update when the spare is built. Once SC commits the split, any query to the server of the failed bucket will

be directed to the new one. This, after being forwarded by the server or the client to SC, as discussed in Section 0. The expected bucket number is necessary in every query to recognize the situation when the server of the failed bucket became a spare in turn and eventually got a new bucket. See [LN95] for further discussion of the algorithm.

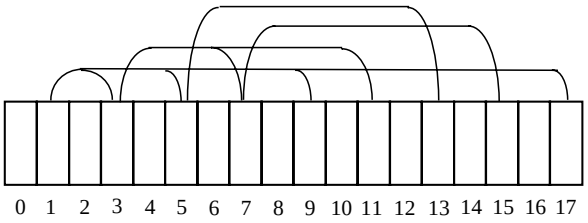

Fig. 2 Children and other descendants of bucket 1 in LH* file with level j = 5

Performance

Load factor

The load factor α of a file is defined as $\alpha = x R_s / b_S N$, where x is the number of records, R_s the record size, b_S the bucket size, and N is the number of buckets in the file. For an LH* file, one has in practice $\alpha \approx$ 70 % - 80 % [LNS93a]. For LH*$_S$ file, assuming the load of α for every segment file, one can show that the approximate value α' of the overall load factor is :

$$\alpha' = \alpha \; k \; / (k + 1) \; - \; O \; [k \, f \,].$$

Assuming k = 4, one ends up with $a' \approx$ 55 - 64 %, i.e., 15-25 % of more storage with respect to the LH* file, as the price for the high-availability.

Inserts

Splits, and forwarding should be infrequent with LH*$_S$, and the failures even more rare. An insert to an LH* file typically costs one message. Hence the typical, and the best, messaging cost of an insert into LH*$_S$ file is (k + 1) messages. Small buckets for segment files make splits, and forwarding more frequent. This increases the average insert cost while building a file, up to 1.5 (k + 1) messages per insert [LN95]. The insert cost increases on the other hand usually by three messages if a segment is sent to a bucket that failed in the meantime hence a spare was created. It may even triple in the worst case, thought about impossible in practice, a discussed in depth in [LN95].

The messaging cost of an insert measured in the number of messages, is at least (k + 1) times higher for an LH*$_S$ file than for an LH* file. Another performance aspect is the volume of data sent over the net. To insert a record of some length l_R, with the key length l_C , one transfers $(l_R - l_c) / k + k \, l_c) \approx l_R / k$ more bytes over the net for an LH*$_S$ file than for an LH* file. For instance for k = 4, it represents only a 25 % increase.

The insert time is determined basically by the CPU time to send-out and receive the message, and by the transfer time. For longer records, over Kbytes, or slower nets, e.g., Ethernet, the transfer time almost entirely dominates [LNS94]. LH*$_S$ typical insert time should then be only 20 - 25% longer than the LH* insert time. For a 10 Mb/s net, and 1 KB record, this leads to the insert time of about 1.25 ms [LNS94]. For faster nets or shorter records, the CPU time begins to dominate. The insert time of LH*$_S$ becomes then closer to (k + 1) /2 times LH* time, as the servers work in parallel, but the client basically serializes the received messages. The exact figures depends on the network speed and topology. For a 1 Gb/s net, 100 Mips CPU, and 1 KB record, the CPU time for an insert into an LH* file may be 51 µs and the transfer time 20 µs, leading 71 µs per insert [LNS94]. The same net would lead to the insert time of 140 µs for the LH*$_S$ file, i.e., two times greater, for, we recall k = 4. Note that this evaluation still neglects the time to segment the record that will add some µs.

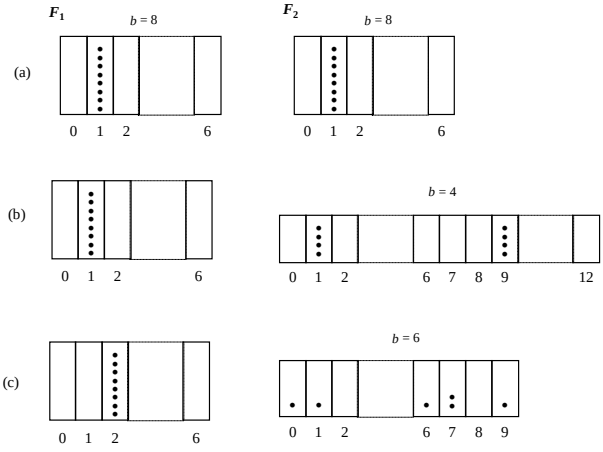

Fig. 3 Types of LH*$_S$ segment files : (a) SA-segments, (b) loosely-coupled SD-segments, and (c) minimally coupled SD-segments

Observe finally that segment files can be on different nets, connected to the client through different controllers. The transfer time decreases accordingly. It can become faster than for LH*.

Key search

The calculus similar to the one for inserts, shows that the typical successful key search costs 2k messages.

An unsuccessful search costs typically 2 messages. In the theoretically worst case without a failure the cost of successful search is 9 k + 1 messages. A bucket failure costs typically an additional message to s_{k+1}, and the reply. It might cost up to 10 additional messages for a very unlucky client. If a failure is discovered during the search, it costs typically 2 additional messages to S_{k+1} and 1 message to SC. With respect to the transfer time, it is about that of LH* assuming $l_c << R_c$, as only k segments are sent back. Hence for large records or slower nets, LH*$_S$ search time is about that of LH*. Otherwise, the successful search time grows towards $2\ k\ t_m$, where t_m is the time to process a message at the client site. In the case of a Gb/s net dealt with for the inserts, the successful search time should be 4 * 56 + 31 = 255 µs, instead of 87 µs for LH* [LNS94]. The unsuccessful search time is clearly about half of it, plus, perhaps the time-out.

Scan search

First component of a scan search price is the cost of query propagation the, let it be m_1 . One has m_1 = 1, if the multicast is used. Otherwise, regardless of the client's image, one has $m_1 = N_S$, where N_S is the total number of buckets of k segment files, as every bucket is reached by exactly one message. The number of rounds is greater when the client's image diverges more, leading to a somehow larger propagation time. Second component, let it be m_2, is the cost of merging all the records, according to Algorithm A4. This cost corresponds to all-to-all bucket messaging, and is for a segment file:

$s = 0.7\ b\ N\ (k - 1)\ / k$,

assuming the load factor $\alpha = 0.7$ on the average. Hence one has :

$m_2 = \Sigma\ s$

and for SA segment files, one has :

$m_2 \approx 0.7\ b\ N\ (k - 1)$.

Finally the third component m_3 depends on the termination algorithm wished. For the probabilistic termination, one depends on the query selectivity and : $0 \le m_3 \le N_S$.

Hence, for SA segment files, one has :

$$0 \le m_3 \le k\ N .$$

For the deterministic termination, one simply has $m_3 = N_S$, as all the buckets must reply.

A practical value of b is $b >> 1$, e.g., b = 1000. Cost m_2 is therefore dominant by orders of magnitude. For k = 4 for instance, and a large file, e.g., N_S = 1000, and, m_2 reaches 2.1M messages. Such messaging has to take at least a few seconds in practice.

Cost m_2 does not exist for LH*. Hence, scans in LH* file are cheaper and faster than in the LH*$_S$ file. This cost is the main price for the high-availability and high-security. If one needs the high-availability only, LH*$_M$ using mirroring allows for without this cost [LN96], but at much larger storage cost. On the other hand, if one segments the records without bit-level scattering, giving up some security, parallel queries may be executed more efficiently, as it will appear below, at the same storage cost as above.

Creating a spare

A spare is created according to Algorithm A5. The messaging cost involves first a few messages between SC and a server where the spare is created. Let this cost be c_1. One can assume $c_1 \approx 2$ in practice. Then RC has to contact servers where the segments used to compute the lost ones could be (Step 3 of Algorithm A5). For SA segments, the cost, let it be c_2 should typically be $c_2 = k$ messages. For SD segments, it can be $k < c_2 \le N_S$. One has $k = N_S$ for the minimally-coupled segment files where the segments for Step 3 has to be searched using parallel queries. Otherwise, one has $c_2 \le pk$, where p is an integer close to $max\ (b_s / b_i)$, where b_s denotes the bucket capacity of the lost bucket, and b_i is the bucket capacity of any other segment file, among the k +1 files.

Next cost component, let it be c_3 corresponds to the join of the segments. One has thus on the average : $c_3 = 0.7\ b\ k$,

as there are on the average as many segments to reconstruct on join servers. Then the reconstructed segments are sent to the spare which leads to the cost component $c_4 = c_3$. Finally, the spare commits to SC, and SC sends the pointer to the spare's parent. All together this leads to the following typical costs :

- for SA files, one has:

$$c_S \approx 2 + k + 1.4\ b\ k + 2,$$

- for SD files, one has:

$$2 + k + 1.4\ b\ k + 2 < c_S \le 2 + max\ (b_s / b_i) + 1.4\ b\ k + 2,$$

- and finally for minimally-coupled segment files, one has :

$$2 + k + 1.4\ b\ k + 2 < c_S \le 2 + N_S + 1.4\ b\ k + 2.$$

Hence the creation of a spare for minimally-coupled segment files can be by far the most expensive.

Multiple bucket failure

It is easy to see that any above discussed LH*$_S$ schema supports a single bucket failure. Resistance to multiple bucket failures depends on whether SA or SD segment files are used. For SA segment files, with bucket capacity of b segments, and no load control, a multiple bucket failure does not create any loss of records, as long as no failed buckets hold segments of the same record. This is an unlikely event. For any two segment files there are indeed only two such buckets.

If this happens anyhow, than one looses αb records, on the average, i.e., 0.7 *b* records in practice.
Loosely-coupled and minimally-coupled SD segment files, increase the probability of data loss in the case of a multiple failure, but decrease the amount of lost data. See [LN95] for the corresponding performance trade-offs.

High-security

Bit-level striping

The bit-level striping as used in an LH*$_S$ file provides naturally the *high-security* in the sense that no record becomes known to an intruder to a site or to a network. For every record *R* of the LH*$_S$ file striped at bit-level, each bucket has one of each *k* bits of *R*. If *l* is the length of *R* in bits, the key non-included, there are $s = l\ (1 - 1/k)$ bits of *R* missing from the any bucket. An intruder to a site has 2^s possibilities to complete a segment to the actual content. This is at least a very long computation in practice.
Next, even an intruder knowledgeable of LH*$_S$ principles, cannot find from the bucket where to find other segments. A bucket in one segment file does have the addressing parameters of other files (except when a scan is in progress). Hence, the intruder would need to search the missing segments anywhere in the multicomputer. One can reasonably expect such a task at least very long in practice.
Finally, LH*$_S$ protects against getting knowledge of the data through the listening on the net. Every message naturally carries only one from every *k* bits of the record. To reconstruct the intercepted segments with the same key, require *k*! trials, assuming the intruder does not know the reconstruction order that is known implicitly only to the client. If this protection is not enough, one can easily scramble the same keys to different values for the transfer in different segments. For instance, the server can add the segment number to the key in the message, to be subtracted by the client. Finally, different segments of a record may come to the client through different sub-networks, making the intrusion through the network listening even more difficult.
Note that SA segment-files are somehow weaken from the high-security point of view than SD files. If an intruder to a bucket finds the addresses of other segments of a record, it knows the addresses of all other records in the bucket. Such correspondence is only partial for loosely coupled SD segment-files, and does not exist for the minimally-coupled files.
Note finally one more nice property of LH*$_S$. Even if an intruder learns the addresses of all the segments of a record at one time, these addresses change when the file scales.

Attribute level striping

LH*$_S$, as discussed above, enhances the high-security at the expense of scan performance. It makes segments meaningless through the scattering at bit level, in order to make data secure against intrusions. To make a scan efficient, through the parallelism, data should in contrast remain possibly entire. If efficient scans are of greater importance than the high-security provided by LH*$_S$ with bit-level striping, one should use the striping at the level of blocks of data.
One choice for LH*$_S$ is the *attribute-level striping*. Each of *k* segments of a record *R* (*c*) contains then *c* and some non-key fields of *R*. Each non-key attribute is entirely in one (and only one) segment. The attributes in a segment do not need to be the consecutive ones in *R*. The parity segment s_{k+1} contains the parity bits for the fault-tolerance. As the segments may be now of different length, s_{k+1} is of the length of the longest one.
The attribute level striping lowers the bucket security level. The intruder disposes of a meaningful part of a record, thought there is still no addresses in the bucket of the rest of *R*. In turn, one may process some scans without first reconstructing the records. This may lead to a substantial performance improvement [LN95].
Attribute-level striping also may lead to a better performance of updates and of the key search. An update to some attribute *A* (*S*) in segment *S*, requires access only to *S* and the parity segment. A search involving only the key and *A* (*S*) requires access to *S* only. LH*$_S$ with the attribute-level striping is more discussed in [LN95].

Related work

The ideas in LH*$_S$ originate in the RAID approach (Redundant Arrays of Inexpensive Disks) [PGK88]. However, LH*$_S$ scatters data over a distributed RAM of servers at a net, instead of a cabinet of disks. Another difference is that the LH*$_S$ stripes at the logical (record, and perhaps attribute) level, instead of physical page (sector) level, using the key as the identifier replicated in each segment. This allows LH*$_S$ to easily scale, unlike the RAID schemes.
The efficiency of the scan search as discussed for LH*$_S$ is not a part of RAID goals. The high-security goal of LH*$_S$ is not a part of RAID idea objectives neither. It follows the *Fragmentation-Redundancy-Scattering* (FRS) proposal for the data management over the networks, [R94]. One postulates in [R94] and its references that the FRS approach is among the most promising ones.
There were other attempts to use striping for network files. An overview of some of the proposed schemes is in [T95a]. Among earliest proposals, was the RADD

(Redundant Arrays of Distributed Disks) schema [SS90]. The RADD schema is also *physical*, striping at page level. It is also static, designed for slow networks, and inefficient for the scans. High-security was not a concern for RADD design.

Between recent high-availability prototypes using a physical schema, there is the Zebra system, [HO95]. Zebra files are not SDDSs, e.g., since a centralized directory is required for the address computation. The system uses striped log-structured files with possibly large segments. It is not efficient for operating on individual records, e.g., in the database application context, [HO95]. In particular there is no provision in Zebra architecture for the scans.

Conclusion

LH*$_S$ appears an attractive SDDS providing the fault-tolerance and high-security of data. Both features are of interest to many applications. The price for new features is a fractional increase to the storage for the file, and some additional messaging, as compared to LH*. The additional cost is moderate, especially when most of file operations are key searches and inserts. Scans may affect the access performance more, especially if bit-level striping is used. One may trade-in some security for the attribute-level striping, improving the access performance.

Further research should concern performance analysis and experiments with various design issues. New ideas for RAID systems may give interesting result when transposed to the multicomputer environment [W96]. Given the commercial importance that multicomputers should have soon, [M96], another interesting alley should be to expand the Windows NT file striping capabilities to the LH*$_S$ capabilities. Finally, one should investigate high-availability variants using striping for other SDDSs, RP* schemes especially [LNS94], [LN96a].

References

[ASS94] Amin, M., Schneider, D.and Singh, V., An Adaptive, Load Balancing Parallel Join Algorithm. 6th International Conference on Management of Data, Bangalore, India, December, 1994.

[C94] Culler, D. NOW: Towards Everyday Supercomputing on a Network of Workstations. *EECS Tech. Rep. UC Berkeley*.

[D93] Devine, R. Design and Implementation of DDH: Distributed Dynamic Hashing. *Int. Conf. on Foundations of Data Organizations, FODO-93. Lecture Notes in Comp. Sc.*, Springer-Verlag (publ.), Oct. 1993.

[G96] Gray, J. Super-Servers: Commodity Computer Clusters Pose a Software Challemge. Microsoft, 1996. http:\\www.research microsoft..com\

[HO95] Hartman J., Ousterhout, J. The Zebra Striped Network File System. ACM Trans. on Comp. Systems. 13, 3, 95, 275-309.

[KW94] Kroll, B., Widmayer, P. Distributing a Search Tree Among a Growing Number of Processors. *ACM-SIGMOD Int. Conf. On Management of Data*, 1994.

[LNS93] Litwin, W. Neimat, M-A., Schneider, D. LH* : Linear Hashing for Distributed Files. *ACM-SIGMOD Intl. Conf. On Management of Data*, 1993.

[LNS93a] Litwin, W., Neimat, M-A., Schneider, D. LH*: A Scalable Distributed Data Structure. (Nov. 1993). To app. in ACM-TODS.

[LNS94] Litwin, W., Neimat, M-A., Schneider, D. RP* : A Family of Order-Preserving Scalable Distributed Data Structures. *20th Intl. Conf on Very Large Data Bases (VLDB)*, 1994.

[LN95] Litwin, W., Neimat, M-A. LH*$_S$: a high-availability and high-security Scalable Distributed Data Structure. U. Paris 9 Technical Report, 1995.

[LN96] Litwin, W., Neimat, M-A. High-Availability LH* Schemes with Mirroring. Intl. Conf. on Cooperating Information Systems. Brussels, (June 1996), IEEE-Press, 1996.

[LN96a] Litwin, W., Neimat. *k*-RP* : a High Performance Multi-attribute Scalable Distributed Data Structure. Intl. Conf. on Par. and Distr. Inf. Sys., IEEE-PDIS 96. IEEE-Press, 1996.

[M96] Microsoft Windows NT Server Cluster Strategy: High Availability and Scalability with Industry-Standard Hardware. A White Paper from the Business Systems Division. Microsoft, 1996.

[PGK88] Patterson, D., Gibson, G., Katz, R., H. A Case for Redundant Arrays of Inexpensive Disks (RAID). ACM-Sigmod, 1988.

[R94] Randel, B. System Dependability. *Future Tendencies in Computer Science, Control, and Applied Mathematics.* Lecture Notes in Computer Science 653, Springer-Verlag, 1994. A. Bensoussan, J. P. Verjus, ed. .21-50.

[SS90] Stonebraker, M, Schloss, G. Distributed RAID - A new multiple copy algorithm. 6th Intl. IEEE Conf. on Data Eng. IEEE Press, 1990, 430-437.

[T95] Tanenbaum, A., S. *Distributed Operating Systems.* Prentice Hall, 1995, 601.

[T95a] Torbjornsen, O. Multi-site Declustering Strategies for Very High Database Service Availabiity. Thesis Norges Techn. Hogskoule. IDT Report 1995.2, 176.

[VBWY94] Vingralek, R., Breitbart, Y., Weikum, G. Distributed File Organization with Scalable Cost/Performance. *ACM-SIGMOD Int. Conf. On Management of Data*, 1994.

[W96] Wilkes, J. & al.. The HP AutoRAID hierarchical storage system. ACM-TCS, 14, 1, 1996.