LH*g : a High-availability Scalable Distributed Data Structure By Record Grouping

Witold Litwin[1], Tore Risch[2]

Abstract

*LH*g is a high-availability variant of the LH* Scalable Distributed Data Structure. An LH*g file scales up with constant key search and insert performance, while surviving any single-site unavailability (failure). High-availability is achieved through record grouping. A group is a logical structure of up to k records, where k is a file parameter. The group members always remain at different sites, regardless of moves due to splits while the file scales up. Every group contains a parity record allowing the reconstruction of an unavailable member. The basic schema may be generalized to support unavailability of any number of sites, at the expense of storage and messaging. Other known high-availability schemes are static, or require more storage, or provide worse search performance.*

Keywords

Scalability, distributed systems, distributed data structures, high-availability, fault tolerance, parallelism, multicomputers

1. Introduction

Multicomputers are collections of autonomous work stations 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]. Many organizations have multicomputers with hundreds or thousands of nodes, whose distributed RAM reaches dozens of GBytes, and disks reach TBytes. Multicomputers offer best price-performance ratio and are potentially capable of computational performance superior to that of supercomputers [T95], [M96]. Research on multicomputers is increasingly popular [C94], [G96].

Multicomputers need new data structures where scalability is vital. The *Scalable Distributed Data Structures* (SDDSs), [LNS93], are intended specifically for multicomputers; other avenues to scalability, especially for parallel supercomputers, are also being explored, [SPW90], [JK93]. An SDDS scales up over the distributed storage, in particular the RAM storage[3]. The first SDDS proposed and becoming well known, was LH* [LNS93], [LNS96], [K98], [R98]. Today, several SDDSs are known. They provide hash based, ordered, or multi-attribute distributed access orders of magnitude faster than to traditional files [D93], [KW94], [LNS94], [VBWY94], [KLR96], [LN96], [TZK96]. SDDS files may also be much larger than centralized files.

It is well known that some applications require the *high-availability* (fault-tolerant) schemes, delivering data despite site failure. Many static high-availability schemes are known, some for multicomputers [T95a], [M96]… The LH*m schemes with record mirroring are first high-availability schemes specifically for scalable files [LN96a]. Another high-availability SDDS scheme, LH*$_S$, stripes every record into *k* stripes (fragments) at different sites, [LN97]. An additional parity stripe is created per record and allows for the recovery of any single stripe.

LH*g proposed below is also a high-availability SDDS. The high-availability is obtained through the new principle of *record grouping*. A *record group* with up to *k* members is a logical construct over LH* file, called *primary* in an LH*g schema. Groups are invisible to the user. They are formed dynamically from records inserted into different buckets (sites). Group members may move through splitting buckets while the file scales up. The scheme guarantees nevertheless that every member remains always in a different bucket. Every group is provided with a *parity* record. The parity records are stored in a separate parity file. One can always recover any single record from all the others in the group and from the parity record. A whole failed bucket is recovered in a hot spare.

A failure-free search in an LH*g file costs no more than this in the LH* file. This includes the parallel search. The storage overhead for parity records for an LH*g file is about 1/*k*. Also, the fault

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

[2] Linköping University, torri@ida.liu.se

[3] To process disk files becomes increasingly cumbersome for database applications. As Jim Gray noted, if the RAM access time were a minute for the CPU, then every disk access would make the application idle for eight days. More prosaically, a simple aggregate function over a typical 4 GByte disk file with the current I/O speed of a few Mb/s easily takes hours.

tolerance can be added, or enhanced, without reorganizing the existing file. The conjunction of all these factors, and of scalability, is unique advantage of LH*g.

The basic LH*g schema generalizes to schemes recovering from many multiple bucket failures, or guaranteeing the recovery from any *n*-bucket failure, for any given $n > 1$. Greater *n* requires higher the storage overhead and recovery costs

The next section presents LH*g. Section 3 discusses performance. Section 4 addresses design variations optimizing selected features. Section 5 positions LH*g within the related work. Section 6 contains the conclusions.

2. LH*g schema

2.1 LH* schema

We first recall the general LH* scheme. The file is stored at *server* computers (nodes), and is used by applications at *client* nodes. Clients are autonomous and can be mobile. The file consists of records (tuples, objects...) identified by (primary) keys, or OIDs, usually noted *c* in what follows. We denote record *R* with key *c* as *R* or *R*(*c*) or simply *c* if its non-key part is unimportant in the context. A client can (key) *search* for *R* given *c*, or may *insert R*, or may *delete R* given *c*. Records are stored in *buckets* with a *capacity* of *b* records, $b >> 1$. We number the buckets of *M*-bucket file as 0,1,2…*M*-1. There is one bucket per server, although different files may share servers.

The physical (network) addresses of the servers are in *allocation tables*. These static or dynamic tables are at the clients and the servers. Addresses of new buckets get propagated according to rules we recall later on.

An LH* file scales up through splits. The splitting and addressing rules follow those of *linear hashing* (LH) [L80]. Every split moves about half of the records in a bucket into a new one. The splits are done in the deterministic order 0…N-1; 0,1…$2N$-1; 0,1…$2^j N$-1,0…; $j = 0,1…$ The value of *N*, usually, $N = 1$, is the initial number of buckets.

The splits are triggered by bucket overflows. A bucket that overflows reports to a dedicated node called the *coordinator* of the LH* file. The coordinator applies a *load control policy* to find whether it should trigger a split. The *split pointer* with value usually denoted *n* indicates the next bucket to split. This bucket is usually different from that reporting the overflow.

Every LH* bucket contains in its header a *bucket level j*. Every initial bucket gets $j = 0$. Access to buckets is calculated through a *linear hashing* function *H* that is dynamically defined from a family of hash functions $h_l : c \rightarrow c \bmod 2^l N$; $l = 0,1…$ Initially $H = h_0$. Then, every split replaces h_j used at bucket *n* with h_{j+1}. The latter assigns new address $n + 2^j N$ to about half of the records in bucket *n*. The coordinator appends to the file the new bucket $n + 2^j N$ and moves there those records. The new bucket gets level $j + 1$.

At any time, for some *i* that we call *file level* ; $i = 0,1…$; all the buckets have level $j = i$, or some buckets have $j = i$ and some $j = i + 1$. The couple (*n*, *i*) constitutes the *file-state data* or the *file-state* in short. The file-state is maintained at the coordinator's node. It defines the *correct address* $a = H(c)$ of key *c* to which *c* should be hashed and where *R* should be. The following LH *addressing algorithm*, computes *a*:

```
(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 have access the file-state. Instead, each client has its own *image* of the file-state denoted (n', i'), initially set to $i' = n' = 0$. These values may vary among clients and may differ from the actual *n* and *i*. The client uses its image and (A1) to calculate the address a' = H (n', i'). It then sends a request to server a'.

It may happen that $a' \neq a$ if the image is incorrect. Hence, any bucket *m* receiving a request first tests whether $m = a$. It can be proven that $m = a$ iff $m = h_j(c)$. If the test fails, the server forwards the request to another server. For the test and forwarding, each LH* servers uses the following algorithm.

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

As for any SDDS, the correct server receiving a forwarded message sends to the client an *Image Adjustment Message* (IAM). For LH*, an IAM contains the *j* value of server a'. It may also contain its physical address, as well as physical addresses of some buckets preceding bucket a'. The value of *n* is not in an IAM, as unknown to the server. The client executes then the *IA-Algorithm*, :

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

As the result of (A3), both i' and n' are closer to the actual values and the same addressing error cannot happen twice.

(A3) makes LH*-images converge rapidly. Usually, O (log M) IAMs to a new client (the worst case for image accuracy) suffice to eliminate the forwarding. In practice, the average key insert cost is one message, and key search cost is two messages, regardless of the file size. A major property of (A2) is also that it delivers every request to the correct address in at most two hops. The worst access performance of an insert into or search in an LH* file is thus four messages, regardless of how much the file scales up.

These figures translate to access times depending on the network and CPU speeds. Experiments with the LH* RAM files on a Windows NT multicomputer using 100 Mbit/s Ethernet, show the key search time of 200 μs [B96], close to the theoretical value of 186 μs in [LNS94]. On a Gbit/s network, key search times should be in general under 100 μs, the CPU speed then becomes a bottleneck. These times are orders of magnitude faster than for disk based files. Note that the distributed RAM of a modern multicomputer can easily hold many GB files that traditionally had to be on disks. Arbitrarily large disk-based LH* files can be created as well.

Another major advantage of LH* files over the centralized ones are the *scan searches*, or *scans* in short, where one searches in parallel every bucket for records with selected non-key values. The client ships a scan using a multicast message whenever the multicast service is available. Otherwise, it uses unicast messages. The client's image may however not show all the buckets. Every bucket should nevertheless get the scan and only once. An efficient algorithm for this purpose is in [LNS96].

Once a scan is sent, the client should determine when all replies arrived. A *probabilistic* termination (protocol) consists of setting a time-out, after the reception of every new record. Only servers with selected records reply. Alternatively, the client sets up for the *deterministic* termination, checking for all the selected records. For LH*, every buckets replies then with at least its address m and its level j_m. The client terminates when it has received, in any order, $m = 0$ and $m = 1$…and $m = 2^i + n$ where $i = \min (j_m)$, and $n = \min (m)$ with $j_m = i$.

Variants of LH* differ with respect to the load control, details of splitting process and the interior bucket structure. In-bucket indexes can be created to speed-up scans through access paths to non-key data. There are LH* schemes without coordinator, as well the already mentioned high-availability variants. The diversification provides best performance for specific applications.

*2.2 LH*g file structure*

An LH*g file F is a pair of LH* files called *primary* and *parity* files F_1 and F_2 with a single coordinator managing the file-state for both files at bucket 0 of F_1. File F_1 has initially $k > 1$ buckets. Every bucket m in F_1, regardless of how much F_1 scales, belongs logically to some *bucket group g*, with k buckets and $g = \text{Int}(m / k)$, Fig. 1a. Records in F_1 are *primary* records. A primary record c contains the *original* record c provided by the application to the LH*g client and record group number $\boldsymbol{g}$ introduced below.

Every bucket m in F_1 has an *insert counter r* with values 0,1,2... The primary record c is *inserted* into bucket m when the original record c comes for storage within bucket m. A record inserted into bucket gets associated with next value of r. A record that *moves* into a new bucket through a split, keeps its original r value.

Every record c in F_1 belongs to a logical *record group* $\boldsymbol{g} = (g, r)$. The value of $\boldsymbol{g}$ is assigned to record c when it is inserted. It remains invariant while the file scales and record c perhaps moves. The value g is the bucket group number of the server where c is inserted. This bucket also assigns the value r. The server adds $\boldsymbol{g}$ to the original record c, Fig. 1b.

For each record group $\boldsymbol{g}$**,** hence for all the records in F_1 currently bearing the record group number $\boldsymbol{g}$**,** a *parity* record exists in F_2. Its key is $\boldsymbol{g}$, and its non-key data are shown in Fig. 1b. Parity buckets are at servers different from those of primary buckets. Parity record $\boldsymbol{g}$ contains the keys $c_1 \ldots c_l$; $l \leq k$; of all the records in group $\boldsymbol{g}$. It also contains the *parity bits* computed from the non-key data of all the records in $\boldsymbol{g}$. There is one parity bit p_i computed for every i-th bits b_i from each non-key part; $i = 0,1\ldots$ The even parity is used in what follows which means that p_i is XOR of b_i's. If non-key data in primary records in $\boldsymbol{g}$ have different lengths, the shorter ones are assumed padded with $b_i = 0$. The content of parity record $\boldsymbol{g}$ suffices to recover any record within group $\boldsymbol{g}$, provided that all other records in group $\boldsymbol{g}$ are available.

Fig. 1a shows how records inserted into buckets forming bucket groups of size $k = 3$ are mapped into parity records, represented with small circles. The number of primary records in any group $\boldsymbol{g}$ is always at most k, and records in $\boldsymbol{g}$ always remain each in a different bucket. A bucket group usually maps to several parity records, shown as circles. Bucket group 0 with buckets 0, 1, 2, has 5 parity records, hence 5 record groups, while bucket group 1 has 4 record groups and parity records. The dashed arcs in Fig. 1a show primary records that splits move. The dashed straight lines illustrate that every such record keeps its original record group number ***g***. *Splitting a primary bucket in an LH*g file does not require therefore to update the parity records*. This property is unique to LH*g at present and has obvious advantages.

It appears from the figure that every member of a record group is inserted into a different bucket. It also appears that members that move continue to remain in separate buckets. For instance, a record in bucket 0 can move into bucket 3, but only records from bucket 1 may move into bucket 4. Later on, other records from bucket 0 may move into bucket 6, but only records from bucket 1 may reach bucket 7, etc. This property is essential, and Proposition 1 will prove it formally. The parity record ***g*** allows indeed the recovery of a single member of ***g*** only. It would not suffice if two members of ***g*** could ever end up in the same lost bucket.

2.3 Building the LH*g file

To store a record R in an LH*g file, the application provides R to an LH*g client. The client sends R to the primary bucket whose address is computed from c according to the LH* principles for F_1. This bucket is the correct bucket m for R, or R reaches *bucket* m after some forwarding. The server of bucket m sets then r to $r := r + 1$, and adds to R the record group key $\boldsymbol{g} = (g, r)$. The resulting primary record R is finally inserted into bucket m.

Bucket m then sends R into F_2, acting as an LH* client and using $\boldsymbol{g}$ as the key. After perhaps some forwarding, R reaches the correct parity bucket m'. If there is no parity record R' (g) in bucket m', then R' is created from R. R' is constituted from $\boldsymbol{g}$, c and from the parity bits $p_i = b_i$. If R' is found, then existing parity bits are updated, through b_i XOR p_i for every b_i in R, including perhaps the padding.

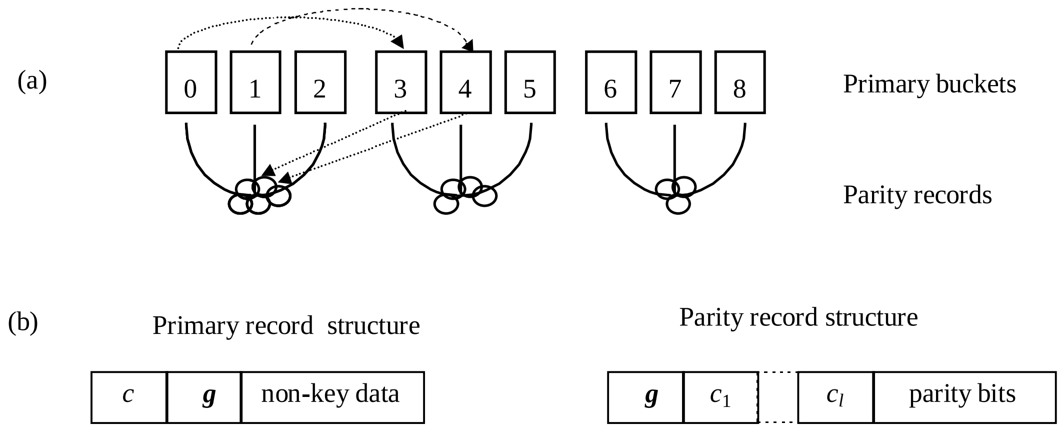

Fig. 1 (a) Mapping of primary records in F_1 into parity records in F_2,
(b) Primary and parity record structures

When a split of F_1 occurs, primary records move with group keys unchanged, as it was discussed. Fig. 2 illustrates more in depth the scalability of the file from Fig. 1. The primary keys c are in italics, the group keys (g, r) are in parentheses, and non-key data are shown as '...' or as 2-bit strings for the records in group (0, 1). The example bucket capacities are $b = 4$ for the primary file, and $b' = 6$ for the parity file. Fig. 2a shows the content of both files just before the first split, of bucket 0. The arrows symbolize the split pointers n, $n = 0$ for each file. The split is triggered by the insert of the record with key $c = 3$. After the split, the split pointer n_1 of F_1 moves to $n_1 = 1$. Record 3 gets $g = 0$ and $r = 5$, despite being immediately moved by the split to bucket 3. File F_2 in Fig. 2a contains parity records for five record groups with $g = 0$ and $r = 1..5$. Group (0,1) has three members with primary keys c_i=12, 16, 59. The parity bits for these records are calculated using even parity. Group (0,2) has 3 members, group (0,3) has 2 members. The remaining groups have one member each.

Fig. 2b shows the situation after the split of primary bucket 0 and three more inserts. The split has created bucket 3, and has set its insert counter r to $r := 0$. The split also moved records with keys $c = 3$, 15, 21 to bucket 3. The counter r of bucket 0 remains unaffected, i.e. with value $r = 5$. The moved records keep their original record group numbers ***g*** with bucket group number $g = 0$, although bucket 3 has $g = 1$. In consequence, the parity records did not to be updated and remain the same.

The inserts that occurred after the split were these of primary records 38, 23 and 33. Record 38 went to primary bucket 2, got $r = 3$, and triggered an update to the already existing parity record (0,3). Key 38 entered parity record (0,3), and the parity bits in the record (not shown) were adjusted. Record 23 was also inserted to bucket 2, got $r = 4$, and caused the update of parity record (0,4). Finally, record 33 was inserted to bucket 3. If it had been inserted before the split, it would have gone to bucket 0, and gotten $g = 0$. Now, it got $g = 1$ as it will be the case of any further inserts to bucket 3. It also got $r = 1$, since the records already in bucket 3 were only those moved from bucket 0. This started a new record group (1,1) and the new parity record (1, 1).

Finally, in Fig. 2c record 42 is inserted to bucket 0. It got $r = 6$, although only two records remained in bucket 0 after the split. It also started a new record group (0,6) and caused the creation of the parity record (0,6). As bucket 0 of F_2 was already full, this caused its split. Note that the split pointer n_2 remained $n_2 = 0$ according to the LH principles, as next split will create bucket 2 from bucket 0. The current split moved all parity records with odd r moved to bucket 1. The buckets of F_1 that are clients to F_2 will be notified of this change as usual for an LH* file, i.e., through the IAMs when they send data incorrectly to bucket 0 of F_2.

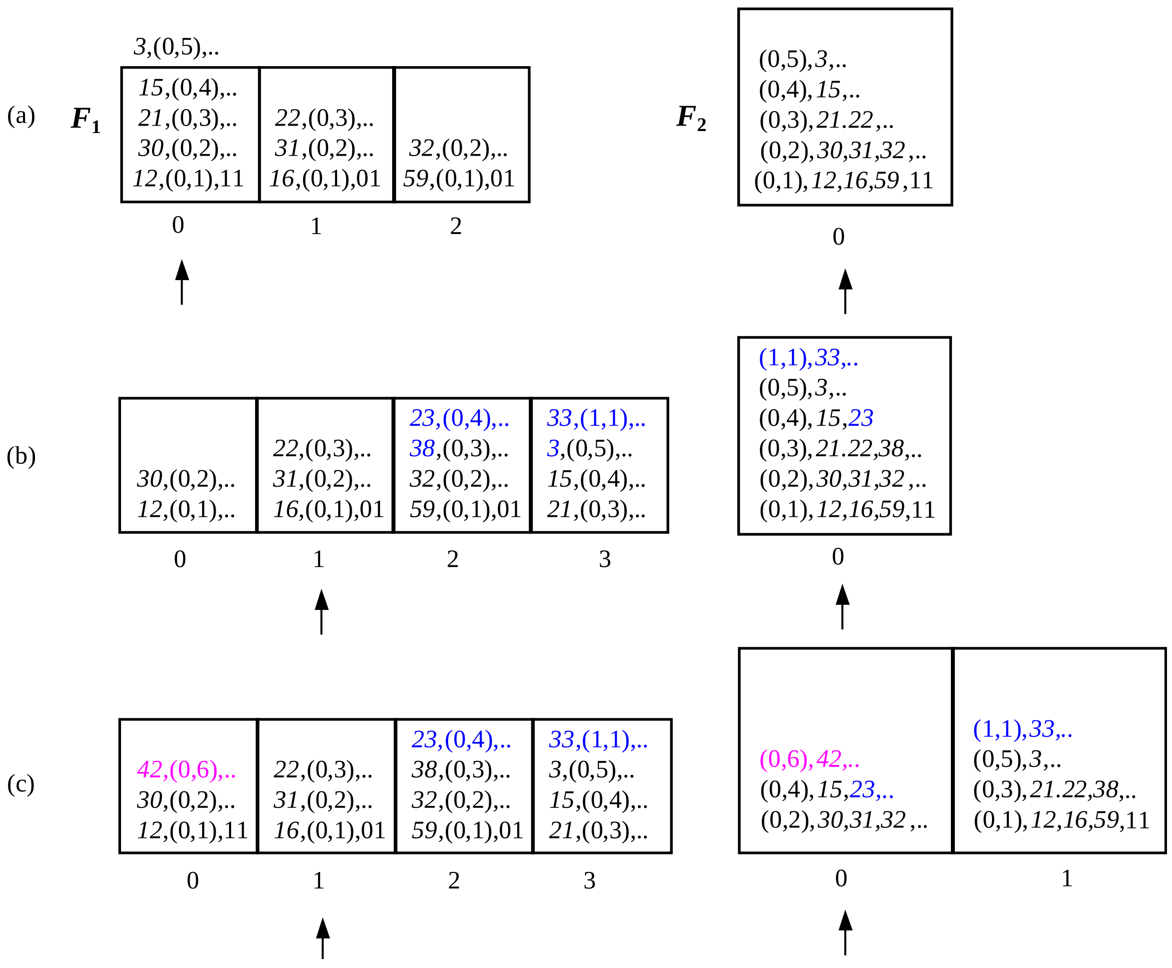

Fig. 2 Evolution of an LH*g file before 1st split (a), and after a few more inserts, (b), (c).

Fig. 1 and Fig 2 illustrate the general behavior of LH*g files that is as follows:

Proposition 1. Regardless of how much an LH*g file scales up, and its primary records move through the buckets of F_1:

1. No record group ever gets more than k members.
2. Any two records in a record group are always mapped to separate buckets.

Proof. 1. A record inserted into bucket m gets the record group key (g_m, r_m) where g_m is the bucket group number of bucket m and r_m is some value of the counter r of bucket m. The counter r of every new bucket is initialized to $r = 0$ and is incremented by one for every insert. It is unaffected by any split of the bucket. Hence, for any r_m, there can be only one record inserted to bucket m with record group (g_m , r_m), although the bucket may contain other records with $r = r_m$, but with $g \neq g_m$, moved there by splits. On the other hand, there can be obviously up to k buckets within any bucket group. Hence, for any g_m and any r_m, there can be up to k records inserted into buckets in bucket group g_m getting the record group key (g_m , r_m). Finally, splits do not change record group keys of the records that move. So, there can be at most k records anywhere in the file with any record group key (g_m , r_m).

2. The principles of LH* imply that any two records inserted to two different buckets m and m' such that bucket m' was not created by a split of bucket m or vice versa, remain in different buckets regardless of moves because of splits. In addition, it follows from the proof of (1) that in an LH*g file, any two primary records c and c' , $c \neq c'$, within the same record group (g, r), are always inserted to different buckets, let them be n and n'. Both properties prove (2) for any n and n' such that bucket n' is not created by a split of bucket n or vice versa. It remains to be proven that otherwise, assuming $n < n'$, a record moved from bucket n by the split creating bucket n' must have the record group number different from that of any record that might be inserted to bucket n'. Assuming that the record that moves has number (g_n , r) for some r and the record inserted into bucket n' has number $(g_{n'} , r')$ for some r', it suffices for (2) to hold that $g_{n'} > g_n$. This is indeed the case, since if j is the level of bucket n when it splits, then the principles of LH* imply that $n' = n + 2^j k$. Hence:

$$g_{n'} = \text{Int}\,(n'/k) = \text{Int}\,((n + 2^j\ k) / k) \geq \text{Int}\,((n + k) / k) = \text{Int}\,(n / k + 1) > \text{Int}\,(n / k) = g_n .$$

2.4 High-availability of LH*g files

We call *high-availability* the continuous capability to deliver stored records despite unavailability (inaccessibility, failure, fault, loss…) of a part of the file storage. For LH*g, we consider the unavailability of data in one or more buckets. An unavailability of a part of a bucket is considered as the unavailability of the entire bucket. A file supporting the unavailability of up to any n buckets will be said to provide *n-availability*. Other facets of availability management, e.g., concerning the recovery of the code running at servers and clients, as, are beyond the scope of the LH*g design. See [T95a] for instance for the discussion of related issues.

It was outlined in Section 2.1 that the LH* scheme allows the file to scale up to any size without access and storage performance deterioration. However, the availability of the file deteriorates progressively. Consider indeed that the probability p that a bucket is available is, e.g., $p = 99\ \%$. The probability P that the whole file F with M buckets is available is $P = p^M$, under the usual assumption that bucket failures are mutually independent. Even if F scales up moderately, let us say to 100 buckets, then P falls to 37 %, i.e., most of the time some data in F are not available. For $M = 1{,}000$, it falls to $P = 0.00004$. Both figures prohibit many applications, as those scanning the file.

The basic LH*g scheme provides 1-availability through the rules that follows. Variants for n-availability are overviewed in Section 4.5.

The unavailability of bucket m, at some physical address s, is detected by a client or a server attempting to access bucket m. It can also be detected by the coordinator, while requesting a split. Finally, it can be self-detected by bucket m, when it restarts itself from an unavailability during which no access to bucket m was requested. The client or the server notifies the coordinator. The coordinator recovers bucket m at some spare server at address $s' \neq s$. The address s' becomes the new physical address for bucket m. If the unavailability occurs during a key search, the coordinator may also attempt to recover only the requested record to complete the search faster. It either delivers then the searched record or determines that no such record existed in the file.

The location change of bucket m is sent to a client only when an IAM is issued. It may happen that an unrelated forwarding occurs and its IAM brings to the client the address s', among other physical addresses [LNS96]. It may also happen that the client is unaware of the new address when it attempts to access bucket m. It sends then the query to address s. If server s is still unavailable, the client resends the message to the coordinator. If server s is up it is either a new hot spare or it carries a new bucket. In both cases, the query is forwarded to address s', as discussed in Section 2.8. The client is finally made aware of the displacement of bucket m by an IAM sent by the server s'.

2.5 Bucket recovery

The bucket to recover may be a primary bucket, or a parity bucket. The recovery of primary bucket 0 includes the file-state data n and i of both files. Self-detected recovery triggers specific actions. The following sections deal with all these cases. We omit easy formal proofs of correctness of (A4) and of (A5) that are in [LR97].

2.5.1 Primary bucket recovery

The coordinator needs to recover every record in the unavailable bucket *m* and the value of counter *r*. It uses for this purpose Algorithm (A4). Through Step 1 of (A4), the coordinator finds all the parity records with keys of records that ever were inserted into bucket *m*, i.e., were there when it failed, or moved away earlier. Step 2 and Step 4 recover the value of counter *r*. Step 3 finds for every record *c* that was in bucket *m* all the records that were in the record group of record *c* at the time of the failure. It then reconstructs every record *c* in the new bucket *m*. Step 2 followed by Step 3 are executed in parallel at every bucket where there was a parity record with a key of a record in bucket *m*.

(A4) Primary bucket recovery

1. The coordinator issues scan Q_1 of file F_2 with deterministic termination. Given the file state (*n, i*) of F_1 and the hot spare *s*, Q_1 requests from every bucket in F_2:

2. The count of parity records with *g* = Int (*m*/*k*), and with key *c* such that $h_l(c) = m$ for some $l \le i + 1$.

3. If some bucket *m*' has in particular a parity record *g* such that for some *c* in record *g*, *m* is the LH* address for *c* in F_1 according to (A1), then bucket *m'* should:
- issue the key search in F_1 for every record *c'* whose key *c'* is in record *g* and is not *c*.
- reconstruct the (missing) record *c* using the parity bits in record *g*
- send record *c* to server *s*.

4. The coordinator sums the counts received in Step 1 and sends the final count *r'* to server *s*, as well as *m* and j_m (determined trivially). Counter *r* in new bucket *m* is set to *r'*, its number to *m*, and level to j_m.

2.5.2 Parity bucket recovery

This is done through Algorithm (A5) where the unavailable bucket *m* is now a parity bucket. Step 1 finds all the primary records whose parity record was in bucket *m*. Step 2 reconstructs bucket *m*.

(A5) Parity bucket recovery

1. The coordinator chooses a hot spare as new recipient of bucket *m*, and instructs it to set up scan Q_2 of file F_1, with deterministic termination. Given the split pointer *n* and file level *i* of F_2, Q_2 requests every record with record group number ***g*** whose address is *m* in F_2.

2. For every set of records with the same *g,* the spare reconstructs the parity record.

2.5.3 File-state recovery

The file-state data *n* and *i* for files F_1 and F_2 have to be recovered into the new primary bucket 0. This is done through Algorithm (A6). It precedes the recovery of primary bucket 0 through Algorithm (A4) that requires the file-state data. Step 1 retrieves bucket numbers and the corresponding levels from the available buckets. Step 2 reconstructs *n* and *i* for F_1 and F_2, under the condition that the split pointer is among the available buckets. Otherwise, the file-state is found through Step 3.

(A6) LH*g file-state recovery

Store in the hot spare *n* and *i* computed as follows:

1. A scan Q_3 with deterministic termination is sent to all available buckets of F_1 and of F_2. Q_3 requests address *m* and bucket level j_m from each bucket *m.*

2. If there is some *m* received such that $j_{m-1} = j_m + 1$, then assign *n* to *n* := *m* and *i* to *i* := j_m.

3. If no such *m* is found assign *M* to the largest *m* retrieved and *i* to *i* := j_1. If $M = k\, 2^{j_1} - 1$, then assign *n* := 0*;* otherwise assign *n* := 1.

Proof. Q_3 will always terminate, since primary bucket 0 is the only one unavailable. For each file, if *n* > 0, then there must be some bucket *m* such that

$$j_{m-1} = j_m + 1.$$

In this case *n* = *m* and $i = j_m$.

If *n* > 1, then Q_3 must retrieve the *m* respecting the above condition. This proves Step 2. Otherwise, one can have *n* = 0 or *n* = 1. Observe that for every LH* file, the largest bucket *m* in the file is bucket *M* such that :

(E1) $$M = n + k\, 2^i - 1.$$

Again, *M* must be among the values of *m* received. Therefore, one can determine the value of *n* using (E1). It follows from the LH* scheme that in case of *n* = 0, all buckets have the same level; hence one can set $i = j_1$. Also in case of *n* = 1, one still has $i = j_1$.

2.5.4 Self-detected recovery

Bucket *m* can self-detect its unavailability through a local test determining data corruption that cannot be recovered locally. It can also detect that it was restarted with the correct data from some temporary unavailability. In both cases, bucket *m* contacts the coordinator before serving any file manipulation. In the first case, it requests the bucket recovery with new bucket *m* at its own address. In the second case, it asks the coordinator whether it is still bucket *m* or whether its unavailability was reported in the meantime and the coordinator has recreated bucket *m* elsewhere. If it was recreated, then the coordinator declares it a hot spare. It informs the bucket accordingly, and sends to it the address of the new bucket *m* The hot spare *m* needs the address of its replacement to update its physical node allocation table.

2.6 Record recovery

The *record recovery* is performed using Algorithm (A7) when a key search requests a record *c* in an unavailable bucket *m*. It provides record *c* to the client or determines that the search would be unsuccessful. It does not restore the record in the file. Its sole purpose is to deliver record *c* during the *degraded* mode, while the recovery of bucket *m* is in progress. The latter has to recover $b >> 1$ records and to create a new bucket. It takes usually much longer than a record recovery.

Step 1 retrieves the parity record from F_2. In Step 2, if no parity record is found, there was no record *c* in the file and the key search terminates unsuccessfully. In Step 3, record *c* is reconstructed if it was the only record in its group. Step 4 addresses the case of more records in the group. The easy proof of correctness of the algorithm is in [LR97].

Algorithm (A7) Record recovery

1. The coordinator sends scan Q_3 with deterministic termination to F_2, requesting parity record ***g*** containing *c*.
2. If the scan is unsuccessful, the search for *c* is terminated unsuccessfully.
3. If *c* is the only key in record *g*, then record *c* is reconstructed from the parity bits in record *g* only.
4. Otherwise, for every other key *c'* in record *g*, the coordinator issues a key search for record *c'*. Then, record *c* is reconstructed from all the received records *c'* and from the parity bits in record *g.*

2.7 Multiple bucket unavailability

Algorithms (A4) - (A7) provide the 1-availability only. The unavailability of two or more buckets in the same bucket group makes indeed the recovery of a record with another record group member also in an unavailable bucket impossible. There are however good cases where some or even all records in failed buckets are recoverable. A primary record sole in record group *(g, r)* is recoverable even if all *k* buckets in bucket group *g* fail. A record group (g, r_1) with less than *k* members may have a single member in all the failed buckets of group *g*, unlike perhaps, unfortunately, another record group (g, r_2). For instance, in Fig 2b, if buckets 2 and 3 fail, the records in groups (0,1), (0,2), (0,5) and (1,1) are recoverable, but not the others.

A record of group (*g*, *r*) in a failed bucket within group *g* may be also not recoverable in presence of multiple failures even if all these failures are outside group *g*. Splits could indeed move a member of its record group to an unavailable bucket.

Algorithms (A4) to (A7) are designed so that a scan, key search, or a request to the coordinator terminate abnormally if they encounter an unavailable bucket. A scan with deterministic termination becomes then unable to get the reply from some buckets it addresses. A key search also blocks on the absence of any reply. On the other hand, if these operations terminate, they provide correct recovery even if there are multiple failures. Hence, multiple failures in an LH*g file cannot lead these algorithms to a silent loss of data.

Further discussion of multiple failures is in Section 4.5.

*2.8 LH*g file manipulations*

An application manipulates an LH*g file as an LH* file. Internally, each manipulation is enhanced for the unavailability management. For an insert of record *c*, as described in Section 2.3, this includes the creation of the record group number ***g***, the message with the primary record *c* to parity bucket *g* and the creation or update of a parity record ***g***. For an update of non-key data *D* of a record within record group ***g*** to new value *D'*, it includes the message to parity bucket *g* with *D'* XOR *D* to update through XOR the parity bits of record ***g***. Every manipulation includes procedures for access to an unavailable bucket. As the general rule, the coordinator handles the manipulations following the detection of an unavailable bucket. In case of an insert, the client sends the new record *c* to the coordinator. The coordinator tests whether *c* already exists in the parity file. If so, it informs the client that the insert would be erroneous, assuming that no duplicates are allowed. If not so, it informs the client that the insert was successful and stores record *c*. In both cases, the client terminates the insert for the application. The coordinator performs asynchronously the appropriate recovery, and completes the insert of record *c*.

As it was outlined in Section 2.4, an LH*g client may send a key search or insert to a former server of a displaced bucket. To resolve the addressing given this constraint, the client always includes in the message the intended bucket number *m*. The rule applies to clients of F_1, and to servers of F_1 acting as clients of F_2.

Assume that the client sends the message to server *s*. If *s* is unavailable, the client resends the message to the coordinator, as it was discussed in Section 2.4. Otherwise, server *s* either (i) carries bucket *m*, or (ii) carries another bucket or (iii) has become a hot spare. In case (i) and (ii), server *s* matches *m* against the bucket number that it carries. If it succeeds, the request is processed as usual through Algorithm (A2). If the matching fails, and in case (iii), the server resends the query to the coordinator which delivers it. It sends the IAM to the client with the new address of bucket *m* in cases (ii) and (iii).

A query can also encounter the unavailability of a forwarding bucket. The sender of any such query, a client or a server, resends the query to the coordinator. The coordinator delivers the query to its correct bucket, bypassing the forwarding through the use of the file-state data and of Algorithm (A1). It always succeeds under the single bucket unavailability assumption.

Under these rules, an LH*g key search for a record in an available bucket does not generate access to F_2. If each forwarding bucket is also available, and the correct bucket is not displaced, the search performs as in the LH* file. Scans translate to parallel searches in F_1. If all primary buckets are available, scans perform also as in the LH* file.

A request for deletion of record *c* from an LH*g file causes the logical or physical deletion of a primary record and of its key from the parity record. The corresponding parity record is updated, if it contains other keys. Otherwise, it is deleted.

A deletion of a record in the record group (*g*, *r*1) decrements the size of the record group and frees value *r*1 of counter *r*. This increases the relative overhead of the storage for the parity record. A logically deleted record consumes also space at the bucket. Nevertheless, it is well known that in scalable files, and modern databases in general, deletions are rare as compared to inserts and updates. Cheap storage creates tendency to store all the historical data. The deterioration of the overall storage efficiency of LH*g due to deletions should thus be usually negligible. If it is not so, one may recollect the storage of the deleted records. Section 4.3 outlines such a variant.

3. Performance

3.1 Storage

Additional storage for F_2 is about 1/*k* of that for F_1, provided that the LH* hash functions h_i hash uniformly (as assumed generally below). The additional storage for F_1 due to *g* values in each record is negligible in practice.

The storage efficiency for an LH* file, as measured by the average load factor, is in practice 70 % if there is no load control, and up to 85 %, with the load control. For the practical values of *k* we will determine later on, e.g., $k \geq 10$, this leads to good average load factor of an LH*g file of about 64 - 77 %.

3.2 Messaging costs in absence of failures

We evaluate the LH*g access performance as usual for an SDDS, i.e., basically through the required number of messages. This measure is network-speed invariant. If a manipulation does not detect a failure, and there was no recovery, *search <u>and scan</u> performance is unaffected by the high-availability features of LH*g*. Since F_1 is an LH* file, and the *g* value in each record does not need to be sent to the client, the performance is simply that of LH* files, as discussed in Section 2.1. If there was a recovery, displaced buckets may induce additional addressing errors, leading to two or three more messages per search or insert, including the IAM. Displaced buckets may also increase the scan cost by a few messages. As failures are rare, the impact of all this on the access performance should be negligible.

An insert to an LH*g file requires typically one more message than for LH*, to update or create the corresponding parity record. This brings the insert cost to typically two messages. In the worst case, obviously almost impossible in practice, it can take up to fourteen messages, assuming two hops per file, and the displacement of all the buckets involved.

Assuming the bucket capacity in F_1 and F_2 equal, the number of splits to create the LH*g file and the corresponding split cost increase typically by the factor 1/*k*.

3.3 Maintainability

We recall that the *maintainability* is a general measure of the service restoration cost [L85]. We evaluate the messaging costs, to get a network invariant formulae. Below, *b* denotes bucket capacity in F_1, and in F_2, again considered equal. *M* is the number of buckets in F_1, hence, there are about M / k buckets in F_2. Finally, we assume no load control, and therefore we assume 0.7 as the average load factor of an LH* file [LNS94].

The record recovery Algorithm (A7), requires first one message from the client to the coordinator, signaling the unavailability. Then, there is the scan of F_2 searching for record *c*. There are about M / k buckets in F_2. A successful search costs then $(M / k) +1$ messages assuming that the scan is sent using a multicast. An unsuccessful search costs $(M / k) + 2$ messages, including the message to the client reporting the negative result.

A successful search for P makes the coordinator sending to F_1 between 1 and k, usually k, key search messages for the records whose keys are in the parity record. The coordinator sends finally the reconstructed record to the client. Hence, the typical total record reconstruction cost is $M / k + 2k + 2$ messages.

In case of a primary bucket recovery using Algorithm (A4), the spare creation results in about $0.7\, b\, (2k - 1) + M / k$ messages. First, Q_3 is sent using a multicast. Then, all M / k buckets of F_2 should reply to the coordinator. There are about $0.7b$ lost records, hence, the same number of parity records. Each bucket with a parity record retrieves then for every record all other members of the group, up to $(k - 1)$, according to Proposition 1. Note that this phase is done in parallel at the corresponding buckets. Finally, each bucket sends all the reconstructed records to the spare. This takes at most $0.7b$ messages, if the reconstructed records are all at different nodes of F_2.

If the failed bucket is a parity bucket, then there is first a multicast message to all buckets in F_1. Then, there are M messages with the selected records or an acknowledgment to provide the deterministic termination. One performs then the reconstruction and sends the records to the spare in one message. This brings the total to $2 + M$ messages.

Finally if the file-state has become unavailable, to recover it using Algorithm (A6) costs $M\,(1 + 1 / k)$ messages.

4. Design variations

4.1 Group size and record recovery

The larger k is, the smaller is the storage for F, relatively to F_1. The downside is that the maintainability costs increase. Nevertheless, as the search performance is independent of k, unlike in striping schemes, larger values of k appear practical, towards $k \geq 10$. Generally, the higher is the file availability[4], the larger should k be.

Larger bucket capacity b' for F_2 leads to more messaging when an F_2 bucket is recovered in the hot spare. In contrast it decreases the number of splits to create F_2, hence is decreases the corresponding messaging. One practical choice is simply $b' = b$.

The deterministic search Q_3 for parity record P, in Algorithm (A7), can be replaced by a time-out search, followed by a deterministic search in the case of no reply. A successful search cost decreases then substantially to 2 messages only, from $M / 2k$ on the average. An unsuccessful search cost becomes $M / k + 1$, instead of M / k, hence remains practically the same.

To recover record c during the degraded mode, Algorithm (A7) uses a scan on parity buckets, as $\boldsymbol{g}$ of record c is unknown. It follows the scan by key searches for primary records required for the reconstruction, using the keys found in the parity record. There are $(k - 1)$ keys at most, hence the latter time is $O\,(k - 1)$ at worst and is constant regardless of the file growth. The scan time is $O\,(b')$, also independent of the file size. It should dominate the recovery time as $b' >> k$. It should be usually acceptable, as it is the typical time of any other scan that applications may request. The degraded mode in the LH*g file is also short lasting, as the bucket recovery is immediately initiated at the spare. If the record recovery time is nevertheless an even more stringent concern, a secondary index internal to each parity bucket can be created, as in general for LH* files. The LH index $(\underline{c}, g)$ shortens then the bucket search time drastically, to about the in-bucket key search time. Key size is usually negligible with respect to the record size, hence the bucket storage overhead should be inconsequential.

4.2 Record group structure

The high-availability can to be added to an LH* file after it was created. This file becomes then file F_1. One may wish then avoid adding g to each original record. This can be achieved. One should create instead a dedicated *link file*, let it be F_3. File F_3 should have records (c, g), containing for each key c its g. If needed, its entire content can be recovered from F_2. Note that the use of link files makes also cheaper the *tuning* of F_2, e.g., choice of a larger k if the bucket availability appears higher than expected. Indeed, records in F_1 do not need to be updated. A further advantage with the link file is that the records in F_2 can have fixed length. The disadvantage is more messages to maintain the link file, and the additional storage for this file.

4.3 Deletions

If many deletions occur, then the LH*g deletion rules can be enhanced to keep the storage overhead from deteriorating in the long run. One has to physically remove then the logically deleted records. Also,

[4]As measured for instance by the already discussed probability P that the entire file is available.

the counter value used by the deleted record should be reused. Otherwise, the average record group size decreases.

Logically deleted records can be removed locally in primary buckets. The implementation dependent details are beyond the scope of LH*g design. The counter values can be reused as follows. Assume that the record *c* with group number (g_1, r_1) is deleted in bucket m_1. First, one (trivially) calculates from g_1 the address m_2 where *c* was originally inserted. If $m_2 \neq m_1$, then the message with r_2 is sent to bucket m_2. Next, in both cases, if bucket m_2 is not empty, then the record *c'* with the largest value of *r*, let it be r_2, gets r_1 instead. The record counter in bucket m_2 is set back by one. Assuming g_2 the group number of bucket m_2, the parity records (g_2, r_2) and (g_1, r_1) are finally updated. That is key *c'* is removed from record (g_2, r_2), and record deleted if it was the last key, or its parity bits are refreshed to include record *c'*. Likewise, key *c'* is added to record (g_1, r_1), and its parity bits are updated accordingly. The whole process costs thus 3 additional messages per deletion at worst. This may appear a fairly cheap price for the improved storage use.

If deletions heavily decrease the number of records, one may also shrink the file. This can be done through the inverse operation to splitting, usually called *bucket merge* for a dynamic data structure. In the case of LH*g, one specific property is that the parity record of a record that moves backwards through the merging primary buckets, may remain unaffected for any such record that was not in the bucket it was originally inserted. The move of the other records in contrast require the change to their record group numbers, to prevent the average size of the record group from decreasing, and related updates to the parity records. Finally, as decreasing number of primary records triggers the decrease to the number of parity records, bucket merge should be applied to the parity buckets as well. Note that the merge of parity buckets occurs asynchronously to that of the primary buckets.

4.4 Overall availability

The discussion in Section 2.7 showed that an LH*g file may not support a multiple unavailability. The worst case occurs when all the *n* unavailable buckets are in the same bucket group. About 0.7*nb* records are then lost.

It is possible to increase the availability *P* of an LH*g file by making recoverable every *n*-bucket failure of buckets in a different group. These are the modifications to the basic schema.

1. When bucket *n* splits, a new group key ***g*** = (*g*, *r*) is attributed to every record *R* that moves to the new bucket *M*. The value of *g* is set to Int (*M* / *k*). The *r* values are set successively in the order of moves to bucket *M*, i.e., the *i*-th *R* gets *r* = *i*. Thus, in difference to the basic algorithm, records moved to a new bucket always get new record group numbers.

2. For every *R*, a new record group *g* is created in F_2, including the parity bits. The key of *R* is also removed from the former group of *R*, and the parity bits are adjusted.

3. The counter *r* of the split bucket is adjusted to the highest value *r* in the record groups remaining in the bucket. A new insert to bucket *n* either reuses any value of counter *r* left free by records that moved, or gets new successive values of *r*.

We call the resulting schema LH*g_1. The new property of LH*g_1 is that a record with bucket group number *g* can be only in a bucket among those forming *g*. Hence, any *n*-bucket unavailability with all *n* buckets in different groups, is recoverable. The worst case remains the same for LH*g and LH*g_1 which are failures of buckets in the same group. Hence, the availability of an LH*g_1 file is strictly higher than that of the LH*g file. The price tag with respect to LH*g are 3 messages to F_2 buckets per record moved, i.e., about 1.5 *b* additional messages per F_1 split on the average.

LH*g_1 has other interesting properties. First, it is no longer necessary to store the bucket group keys in records of F_1. Also, the bucket reconstruction algorithms may become more efficient. All the primary records needed for the reconstruction are indeed in the same bucket group. One can thus apply bulk (stream) transfers to bring entirely the corresponding (*k* - 1) buckets to the hot spare. Next, there is no need to compute the lost value of *r* using a scan of F_2, unlike for LH*g in Algorithm (A4). It suffices to find the maximal *r* in the recovered records, which is a cheaper operation. Higher availability, and all these advantages, may justify for many applications, the increased primary bucket split cost of the LH*g_1 file.

4.5 **LH*g schemes with *n-availability***

Some commercial applications, especially with large files, already provide 2-bucket availability [I97]. LH*g schema can be generalized to *n*-bucket availability with $n > 1$ as well. One approach is *multigrouping* that is to make every primary record belonging to *n* different record groups, [LR97]. These groups are formed so that they intersect at a most one record. This property already characterized so-called *multidimensional* schemas, e.g., those overviewed in [SS90] or [H&al94]. In these schemes, stripes are organized into an *n*-dimensional cube. However, the scalability of LH*g files, with the infinite bucket address space, precludes any trivial transposition of such schemes. One has to ensure in particular that no split ever brings records in the same record group into the same bucket. One solution is to structure the bucket address space into an infinite series of squares *k* x *k* along the diagonal in 2-dimensional (*x*, *y*) space. Groups of size *k* are formed along *x* and *y*-axis and along the diagonal, if $n > 2$ is required. See [LR97], and [LMR98] for details.

It can be observed that the problem of moving records is easier to solve in practice for LH*g_1. One attractive property, new for high-availability schemes, that appears is that *n* may itself gradually scale with the file. This is necessary to preserve the overall availability of a scalable file from deteriorating while the file grows [LMR98].

Another avenue yet to explore is to keep each record in a single group, but to expand the parity schema to tolerate multiple failures within the group. Again the scalability requirement precludes a trivial application of earlier well-known proposals, e.g., in [H&al94] or [ABC97]. At present, Reed-Salomon codes, already in commercial use for data storage systems where scalability is not yet a concern, [I97], appear the best candidate. One potential hurdle, when scalability is taken into the account, is the efficiency of related operations in a tower of Galois fields [LS98].

5. Related work

High-availability schemes for the centralized environment are known for many years. They are based on mirroring or striping, e.g., RAID schemes [PGK88]. No scheme using the concept of record grouping was known.

Unlike the grouping, basic striping schemes usually deteriorate search performance, [W96]. Insert performance is also affected. To overcome this drawback, many variants were proposed, e.g., combining mirroring and striping [W96]. Progressively, the high-availability in distributed environment became also important, [M96]. Overviews of the historical evolution of the domain, and of many schemes, are in [HO95] and [T95a].

Known high-availability schemes are classified into *physical* or *logical* schemes [T95a]. The physical schemes provide high-availability through some redundancy of physical units of storage: bits, bytes, sectors, segments or pages. RAID schemes are among best known physical schemes. The logical schemes manipulate data elements identified by a key or an OID and usually structured: records, tuples, relation fragments or replicas, objects... The additional semantics seems to make logical schemes intrinsically more efficient for a distributed environment [T95a]. Up till now, such schemes are used mainly for parallel or distributed database systems. The LH*g schema is a logical scheme.

When a node failure is detected, some existing schemes redistribute data over available nodes (e.g., [HST91], see also [T95a] for more references). In contrast, others replace the unavailable node with a spare node where the unavailable data are reconstructed. The latter principle seems more efficient for high-availability, [T95]. One reason is that chances for a 2-node failure are typically greatly reduced. It also appears the best approach when distributed data need to scale [LN96a]. LH*g falls into this class of schemes as well.

Among earliest research investigations of schemes for the distributed environment, was the RADD (Redundant Arrays of Distributed Disks) schema [SS90]. This was an application of the RAID-5 schema to disks distributed over some sites. Unlike LH*g, the RADD schema was physical and static, designed for slow networks, and was rather inefficient for parallel selections since it used striping. A similarity to multigrouping in LH*g for *n*-unavailability was the suggestion of a multidimensional generalization of RADD, into so-called *nD*-RADD schemes to support *n*-site availability [SS90].

Among recent high-availability prototypes using a physical schema, there is the Zebra system, [HO95]. Zebra files are not SDDSs since, e.g., 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 context, [HO95]. In particular there is no provision in the Zebra architecture for parallel scans.

The logical schemes proposed for distributed or parallel database systems, e.g., Gamma, Tandem or Teradata machines, were based on static partitioning of relations into fragments, using the locality of reference, key ranges, static hashing, etc.. High-availability is provided by the mirroring of the entire relations, or of fragments, or of the tuples. The replica are placed on different sites or nodes of a multiprocessor machine, using various allocation schemes often called *declustering schemes*, [HD90], [T95a]. All known declustering schemes target a single machine or a cluster of a few machines, and are not designed for scalability as SDDSs in general and LH*g in particular. They basically, or exclusively, support 1-bucket (site, node) availability. The use of replica leads these schemes to higher storage requirements than for LH*g, basically *k* times higher.

Specifically for SDDSs, in [LN96a] one discusses high-availability schemes for LH* using mirroring, LH*m. These schemes also require about *k* times more additional storage, than LH*g for a practical *k* value[5]. In turn, they may make the spare production and record reconstruction faster, i.e., their maintainability costs may be lower. They may finally allow for a higher search throughput, as searches may be distributed over *n* mirrors.

In [LN97], one defines high-availability scheme using striping, called LH*$_S$. LH*$_S$ provides 1-bucket availability. It also increases data security. An intruder to a site may see only a stripe of a record, usually meaningless, since containing only one of every *k* >> 1 bits of each record.

The storage requirements of LH*$_S$ are about those of LH*g and decrease for larger *k*. The value of *k* affects, however, search performance for LH*$_S$, while it does not for LH*g. Hence, larger values of *k* can be chosen in practice for an LH*g file leading to smaller storage. Search performance differences of LH*$_S$ with respect to LH*g are affected especially for scans. To evaluate the search clauses, one may need to reconstruct somewhere every record from its segments, as for any schema based on striping. Maintainability of LH*$_S$ is somehow more efficient than that of LH*g since less data have to be transferred over the net for a bucket or record recovery.

	Search	Insert	Storage	H-availab.	Maintainab.	Throughp.	Security
LH*	1	1	1	N	N	2	2
LH*m	1	2	4	1	1	1	4
LH*s	2	4	3	2	2	4	1
LH*g	**1**	**3**	**2**	**1**	**3**	**3**	**3**

Table 1 Ranking the high-availability LH* schemes

Higher data security remains an inherent advantage of LH*$_S$ over all other schemes. The overall security of LH*g and LH*m is in addition somehow lower than that of LH*. One can indeed break the security of a site looking into other sites. For LH*m, these are *n* mirror sites with entire records. For LH*g, these are buckets from which the record could be reconstructed. Note that this requires breaking the security of *k* sites usually for LH*g, while to break into a single mirror site may suffice for LH*m. In this sense, LH*g is intrinsically more secure than LH*m.

Table 1 ranks the discussed algorithms, including LH*, along their various features discussed above. LH* does not provide high-availability, so it has no rank. LH*g is ranked assuming *k* > 1, to distinguish it from LH*m. If one sums up the basic factors, i.e., the search, insert and storage ranks, LH*g overall rank of 6, makes it best among the high-availability schemes, before LH*m with the rank 7 and LH*$_S$ with the rank 9. Application specific requirements should ultimately help to choose the most adequate scheme.

6. Conclusion

LH*g provides the high-availability to scalable files at no additional search cost and with moderate storage overhead. It should be attractive to applications where both the high-availability and the search performance, in particular of scans, are of prime importance. Modern database systems make the scalability, high-availability and parallelism their major features. LH*g should be particularly useful in this context.

Future work should address various design issues of LH*g. The scheme should be implemented in popular environments. The prototype implementation in Java confirms that it is simple to set up and provides expected performance [L97]. Experiments with actual applications should follow. For the database use, transaction and concurrency management should be addressed. The idea of high-availability through grouping should finally be ported to other SDDS schemes, e.g., RP* for ordered files [LNS94].

Acknowledgments. *The comments of Jonas Karlsson, Ronny Lindberg and of Thomas Schwarz were very helpful, as well as those of anonymous reviewers.*

References

[ABC97] Alvarez, G., Burkhard, W., Cristian, F. Tolerating Multiple-Failures in RAID Architecture with Optimal Storage and Uniform Declustering. Intl. Symp. On Comp. Arch., ISCA-97, 1997.

[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...

[B96] Bennour, F. An Architecture for an SDDS File Management System under Windows NT. DEA Tech. Report. U. Paris 9, 1996, 47. *In French.*

[5]Note that LH*m can be seen as a special case of LH*g , namely when *k* is set to *k* = 1.

[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\

[HD90] Hsio, D. DeWitt, D. Chained declustering: A new availability strategy for multiprocessor database machine. 6-th. Intl. IEEE Conf. on Data Eng. IEEE Press, 1990.

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

[HST91] Hvasshovd et &. A continuously available and highly scalable transaction server. 4th Intl. Workshop on High Perf. Trans. Systems, 1991.

[I97] IBM RAMAC Virtual Array. IBM Redbook SG-24-4951-00., 1997.

[JK93] Johnson, T. and P. Krishna. Lazy Updates for Distributed Search Structure. *ACM-SIGMOD Int. Conf. On Management of Data*, 1993.

[KLR96] Karlsson, J. Litwin, W. Risch, T. LH*lh: A Scalable High Performance Data Structure for Switched Multicomputers. Intl. Conf. on Extending Database Technology, EDBT-96, Avignon, March 1996.

[K98] Knuth, D. THE ART OF COMPUTER PROGRAMMING. Vol. 3 Sorting and Searching. 2nd Ed. Addison-Wesley, 1998, 780.

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

[L85] Laprie, J.-C. Dependable Computing and Fault Tolerance: Concepts and Terminology. 15-th Intl. Symp. on Fault-Tolerant Comp. Soc. Press, 2-11.

[L97] Lindberg., R. A Java Implementation of a Highly Available Scalable and Distributed Data Structure LH*g. Master Th. LiTH-IDA-Ex-97/65. U. Linkoping, 1997, 62.

[LMR98] Litwin, W., Menon, J., Risch, T., LH* schemes with Scalable Availability. IBM Almaden Research Rep. RJ 10121 (91937), May. 1998. (subm. for publ.).

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

[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.

[LNS96] Litwin, W., Neimat, M-A., Schneider, D. LH*: A Scalable Distributed Data Structure. ACM Trans. On Database Systems, ACM-TODS, (Dec, 1996).

[LN96] Litwin, W., Neimat. *k*-RP*$_N$: a High Performance Multi-attribute Scalable Distributed Data Structure**.** IEEE Intl. Conf on Par. and Distr. Inf. Syst., 1996.

[LN96a] Litwin, W., Neimat, M-A. High-Availability LH* Schemes with Mirroring. Intl. Conf. on Cooperating Information Systems. Brussels, (June 1996), IEEE-Press, 1996.

[LN97] W. Litwin, M-A Neimat, G.Levy, S. Ndiaye, T. Seck. LH*$_S$: a high-availability and high-security Scalable Distributed Data Structure. IEEE-Res. Issues in Data Eng. (RIDE-97), 1997.

[LR97] Litwin, W., Risch, T. LH*g : a High-availability Scalable Distributed Data Structure by Record Grouping. Res. Rep. U. Paris 9 & U. Linkoping, (Apr., 1997).

[LS98] Litwin, W. Schwarz, Th. Scalable high-availability LH* schemes with Reed-Salomon codes. CERIA Res. Rep. U. Paris 9, (Oct. 1998).

[LT96] Litwin, W. Tore Risch, T. LH*g : a High-availability Scalable Distributed Data Structure by Record Grouping. Tech. Report. U. Paris 9, U. Linkoping. (May, 1997).

[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.

[R98] Ramakrishnan, K. Database Management Systems. McGraw Hill, 1998.

[SPW90] Severance, C., Pramanik, S. Wolberg, P. Distributed linear hashing and parallel projection in main memory databases. VLDB-90.

[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.

[TZK96] Tung, S, Zha, H, Kefe, T. Concurrent Scalable Distributed Data Structures. ISCA Intl. Conf. on Parallel and Distributed Computing Systems. K. Yetongnon and S. Harini, (ed.) Dijon, (Sept., 1996). 131-136.

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

[U94] Ullman, J. New Frontiers in Database System Research. *Future Tendencies in Computer Science, Control, and Applied Mathematics.* Lecture Notes in Computer Science 653, Springer-Verlag, 1994. A. Bensoussan, J. P. Verjus, ed. 87-101.

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