LH*g : A High-Availability Scalable Distributed Data Structure By Record Grouping

Witold Litwin and Tore Risch, *Member*, *IEEE*

Abstract—LH*g is a high-availability extension 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). We achieve high-availability through a new principle of record grouping. A group is a logical structure of up to k records, where k is a file parameter. Every group contains a parity record allowing for the reconstruction of an unavailable member. The basic scheme may be generalized to support the 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.

Index Terms—Scalability, distributed systems, distributed data structures, high-availability, fault tolerance, parallelism, multicomputers.

1 Introduction

Multicomputers are collections of autonomous workstations or PCs on a network (*network multicomputers*), or of share-nothing processors with local storage linked through a high-speed network or bus (*switched multicomputers*) [25]. Multicomputers offer the best price-performance ratio and have the potential of computational performance superior to that of supercomputers [3], [5]. They require new data structures where scalability is vital. The *Scalable Distributed Data Structures* (SDDSs) [15] are designed specifically for multicomputers. An SDDS scales up through splits of its buckets stored in a distributed storage, RAM storage[1] in particular. The LH* was the first SDDS [15], [17], [2] and many other followed, [4], [11], [16], [28], [9], [18], [27] [10]. They allow for hash-based, ordered, or multiattribute distributed files that can be much faster and larger than traditional ones. They also allow for efficient parallel scans of the files.

Some applications require *high-availability* (fault-tolerance), able to deliver all the data despite the unavailability of some servers, and to recover these servers transparently. $\mathrm{LH}^*_{\mathrm{m}}$ featuring record mirroring is the first high-availability scheme designed specifically for scalable files [19]. Another high-availability SDDS scheme, $\mathrm{LH}^*_{\mathrm{S}}$, stripes every record into k stripes (fragments) at different sites [20]. An additional parity stripe is created per record that allows for the recovery of any single stripe.

Here, we propose a new high-availability SDDS we call LH*g. We obtain the high-availability through the new principle of *record grouping*. Every application record is in a distinct *record group* formed from up to k members and provided with a *parity* record. Any single record can be recovered from all the others in the group and from the parity record. An entire unavailable bucket is recovered in a hot spare.

1. To process disk files becomes increasingly cumbersome for database applications. As Gray noted, if the RAM access time were a minute for 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.

- *W. Litwin is with Université Paris Dauphine, Place du Maréchal de Lattre de Tassigny, 75775 Paris cedex 16, France. E-mail: Witold.Litwin@dauphine.fr.*
- *T. Risch is with Uppsala University, Box 256, 751 05 Uppsala, Sweden. E-mail: Tore.Risch@dis.uu.se.*

Manuscript received 3 June 1997; revised 14 Jan. 1999; accepted 2 Apr. 2001. For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 105149.

A normal (failure-free) search in an LH*g file costs, in practice, the same as in an LH* file. The storage overhead for parity records for an LH*g file is about $1/k$. The combination of these properties with the scalability is an advantage unique to LH*g scheme at present.

We designed the basic LH*g scheme so that splits do not access the parity records to minimize the split cost in the normal mode to that of LH*. The consequence on the degraded mode, when unavailability occurs, is that one rarely recovers from a multiple bucket failure and that the record recovery uses a parallel search of the parity buckets. A variant reorganizes the parity records during the split and trades the higher runtime split cost for a more frequent recovery from a multiple failure, and cheaper record recovery with the access to one parity bucket only. Other variants further enhance the recoverability of multiple failures, or trade-off other performance factors and design issues.

The next section presents LH*g. Section 3 discusses performance. Section 4 addresses the variants' design. Section 5 positions LH*g within the related work and Section 6 concludes.

2 LH*g Scheme

2.1 LH* Overview

An LH*g scheme is a generalization of the LH* scheme. An LH* file resides at *server* computers (sites) and is manipulated from *client* sites. The file consists of records identified each by a primary key, called c below. Records are stored in *buckets* with a *capacity* of b records, $b \gg 1$. We number the buckets $0, 1, 2 \ldots M-1$ for a file with M buckets. Basically, there is one bucket of a file per server.

An LH* file scales up through splits. A split moves about half of the records in a bucket into a newly created bucket. The splits are done in the deterministic order $0 \ldots N-1$; $0, 1 \ldots 2N-1$; $0, 1 \ldots 2^j N - 1, 0 \ldots$; $j = 0, 1 \ldots$. The value of N is the initial number of buckets.

A split responds to a bucket overflow. A bucket that overflows reports to a dedicated node called the *coordinator* of the LH* file. The coordinator chooses to split the bucket pointed to by the sequentially advancing *split pointer*. We usually call its value n. Bucket n is usually not the one 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 $h_l : c \rightarrow c \bmod 2^l\, N$; $l = 0, 1 \ldots$. 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 those records there. The new bucket gets level $j + 1$.

We define the *file level* i $(i = 0, 1, 2 \ldots)$ to be the minimum bucket level. All buckets have bucket levels $j = i$ or for some $j = i + 1$. The pair (n, i) constitutes the *file-state*. The coordinator maintains the file-state. The current value of (n, i) defines the *correct address* $a = H(c)$ for record $R(c)$ as:

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

LH* clients do not have access to the file-state to avoid hot spots. Instead, each client has its own approximate *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. To find the address a' of $R(c)$, the client applies (A1) to c through its image and sends its request to bucket a'. It may happen that $a' \neq a$.

Any bucket m receiving a request first tests whether $m = a$. It has been proven that $m = a$ iff $m = h_j(c)$. If the test fails, the server forwards the request to another server. There, the test and forwarding may repeat once more there. A major property of LH* is that this is the worst case.

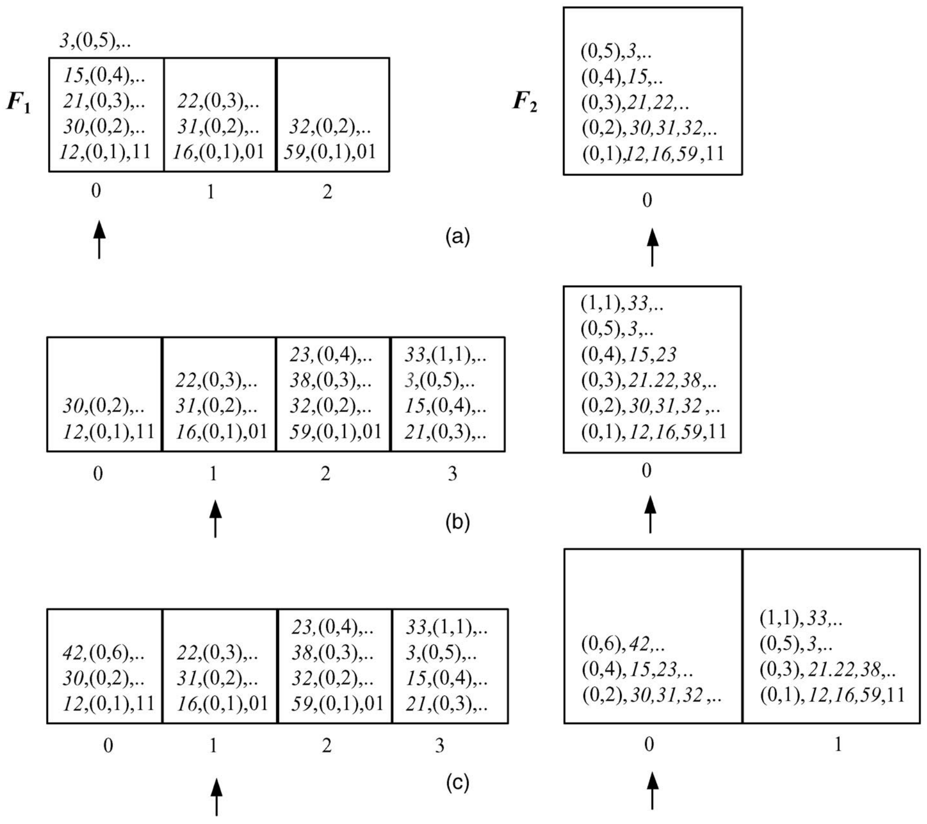

Fig. 1. Structure and evolution of an LH*g file.

The correct bucket a receiving a forwarded message sends to the client an *Image Adjustment Message* (IAM). An LH* IAM contains the j value of bucket a'. It may also contain its physical address, as well as physical addresses of some buckets preceding bucket a'. Then, the client executes the *IA-Algorithm*. As a result, the same addressing error cannot happen twice and i' and n' become closer or equal to the actual values.

An LH* file also supports *scans*, searching in parallel every bucket for selected nonkey values. The client ships a scan with a multicast message, if available, and with unicast messages otherwise. The client's image may not show all the buckets. There is therefore an LH* algorithm accessing each of M buckets exactly once. Likewise, there is an algorithm for *deterministic termination*, checking that all the M buckets replied. LH* also offers scans with *probabilistic* termination which we do not consider further.

2.2 LH*g File Structure

An LH*g file F is, in essence, a pair of LH* files at different sets of servers, the *primary* file F_1 and the *parity* file F_2. One coordinator manages the file-states for both files at bucket 0 of F_1. File F_1 has initially $k > 1$ buckets. Every bucket m in F_1 belongs logically to some *bucket group*, with k buckets and the unique bucket group number $g = \mathrm{Int}(m/k)$. Records in F_1 are *primary* records. A primary record c contains a data record c and a record group number $\boldsymbol{g}$ introduced below.

Every bucket m in F_1 has an *insert counter* r with values $0, 1, 2\ldots$ incremented for each insert. Every primary record belongs to a *record group* $\boldsymbol{g} = (g, r)$. The record receives its $\boldsymbol{g}$ value at the time of insert and the $\boldsymbol{g}$ value remains constant while the file scales and splits move the record. For each record group $\boldsymbol{g}$, a distinct *parity* record with $\boldsymbol{g}$ value as (unique) key exists in F_2. The nonkey data of parity record $\boldsymbol{g}$ have two fields. One contains the list of the keys $c_1 \ldots c_l$; $l \leq k$; of all the records in group $\boldsymbol{g}$ (a record group can have at most k members). The other contains the *parity bits*, computed from the nonkey bits of the primary records in $\boldsymbol{g}$. A parity bit p_i is the XOR of all the ith bits b_i from the nonkey parts of the records. We pad with bits $b_i = 0$ if primary records are of variable length. The content of parity record $\boldsymbol{g}$ suffices to recover any record within group $\boldsymbol{g}$, provided the availability of all the other records in group $\boldsymbol{g}$.

For any record group $\boldsymbol{g}$, each record in $\boldsymbol{g}$ is inserted in a bucket of group g. Later, the record may move. As group keys $\boldsymbol{g}$ of records that move remain invariant, the moves neither affect the record group structure nor the parity records. In this way, *splitting a primary bucket does not require accesses to the parity records*. This highly advantageous property is unique to LH*g at present.

Fig. 1 illustrates these principles. The primary keys are in italics, the group keys are in parentheses, and the parity bits are shown as "..." or as 2-bit strings for the records in group (0, 1). The primary bucket capacity is $b = 4$ and that of the parity bucket is $b' = 6$. The group size is $k = 3$. Fig. 1a shows the file before the first split of bucket 0, triggered by the insert of the overflow record with key $c = 3$. Record 3 gets $\boldsymbol{g} = (0, 5)$. The arrows symbolize the split pointer for each file. File F_2 contains five parity records. Group (0,1) has three members with primary keys $c_i = 12, 16, 59$. The figure shows the first two parity bits in this parity record. Group (0,2) has three members and group (0,3) has two members. The other two groups have one member each.

Fig. 1b shows the file after the split of primary bucket 0 and three more inserts. The split has created bucket 3 and moved records 3, 15, and 21 there. The counter r of bucket 0 remains unaffected with the value $r = 5$. The moved records keep their original record group numbers $\boldsymbol{g}$ with bucket group number $g = 0$, although bucket 3 has $g = 1$. In consequence, the move did not update any parity record.

The inserts that occurred after the split were those 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 got $r = 4$ and caused the update of parity record (0,4). Finally, record 33 entered bucket 3. If it had been inserted before the split, it would have gone to bucket 0 and gotten $g = 0$. Instead, it got $g = 1$ as will be the case of any further inserts to bucket 3. It also got $r = 1$ since the only records already in bucket 3 were those moved from bucket 0. This started a new record group (1,1) and the new parity record (1, 1).

Finally, in Fig. 1c record 42 enters 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. The split moves the parity records with odd r to bucket 1. The split pointer remained equal to 0 according to the LH principles. A bucket of F_1 that sends later data incorrectly to bucket 0 of F_2 will receive the IAM of an LH* file.

The scheme has two key properties, lengthily proven in [21]. First, regardless of the number of splits that could move the records and of new inserts, a record group has at most k members. Second, these members always remain in separate buckets. For instance, for $k = 3$, records from bucket 0 can move progressively into buckets $3, 6, 9\ldots$ but records initially in bucket 1 may move into buckets $4, 7, 10\ldots$ only. These properties guarantee that no two members of $\boldsymbol{g}$ ever end up in the same lost bucket, the key to the high-availability of the scheme. The parity calculus above allows the recovery of a single member of $\boldsymbol{g}$ only and does not support multiple failures.

2.3 File Manipulation

An LH*g file appears to the application as an LH* file. Internally, any LH*g manipulation starts in *normal* (failure-free) mode. If it encounters an unavailable server, then it enters the *degraded* mode.

2.3.1 Normal Mode

An application provides a record R to insert with key c to an LH*g client that sends it out according to its image. R eventually reaches its primary bucket m. If there is any forwarding, it results from the LH* addressing rules or is an additional hop resulting from the case of a *displaced bucket* which we introduce soon. At bucket m, R is stored with the record group key $\boldsymbol{g} = (g, r)$ and the insert counter r is set to $r := r + 1$, as in Fig 1. This may cause a split of primary bucket n, in which case, a primary record that moves, keeps its group key unchanged.

Bucket m also resends R to F_2, acting as an LH* client and using $\boldsymbol{g}$ as the key. Eventually, R reaches the correct parity bucket m'. If there is no parity record R' with key $\boldsymbol{g}$ in bucket m', then R' is created from R. R' is constituted from $\boldsymbol{g}$, c, and from the parity bits p_i initialized to b_i. This may cause the split of a parity bucket and later trigger IAMs to the primary buckets. If an R' is found, then the existing parity bits are updated through XORing, as we have seen.

Key and scan searches execute as for LH*, except perhaps for the encountering of displaced buckets. No search accesses F_2. Next, an update of nonkey data D of a record within record group $\boldsymbol{g}$ to new value D', changes the parity record $\boldsymbol{g}$ whose parity bits are now bitwise D' XOR D. Finally, a deletion of a primary record R with key c causes its logical or physical deletion in its primary bucket. This may trigger the *merge* operation of LH* scheme, inverse to a split. The values of r freed by the deletions either remain unused, or are reused for the next inserts to the bucket, which is better for the storage efficiency of F_2, [21]. The bucket also sends R to the parity bucket with parity record $\boldsymbol{g}$. Parity record $\boldsymbol{g}$ that also contains keys other than c is updated accordingly. That is, key c is removed and the incoming bits are XORed with the existing ones. Otherwise, record $\boldsymbol{g}$ is deleted. Notice that deletions are rare in scalable files and modern databases in general, because of the tendency to store all the historical data.

2.4 Degraded Mode

If bucket m at some site s is unavailable, a client or a server may detect its unavailability while attempting to access it. The server of bucket m may also self-diagnose an unavailability. In all the cases, the coordinator is notified. The coordinator may also detect the unavailability while requesting the split.

If the notification comes from the client, it contains the request. The coordinator first checks its file-state data whether bucket m is the correct one for the request. If not, it delivers the request to the correct bucket, unless that one is unavailable as well. The latter case is unrecoverable for the basic LH*g scheme.

An insert in the degraded mode terminates for a client when the coordinator receives the record. The recovery occurs asynchronously. A key search triggers the *record recovery,* which delivers the requested record or determines that it is not in the file. Any unavailability detection leads eventually to the recovery of the full bucket. The bucket to recover may be a primary, or a parity one. All the records of the bucket, let it be at server s, are recovered at some *spare* server with address $s' \neq s$. The displacement later triggers the propagation of the new address through IAMs. The recovery of primary bucket 0 includes the file-state data.

2.4.1 Record Recovery

To recover record c, the coordinator performs a scan of F_2 for every record $\boldsymbol{g}$ containing c. If no record is found, the search for c is declared unsuccessful to the client. If c is the sole key in record $\boldsymbol{g}$, the parity bits constitute the nonkey part of record c. If there are other keys in record $\boldsymbol{g}$, the corresponding records are found and used for the recovery through the XORing of their parity bits with those in record $\boldsymbol{g}$.

2.4.2 Primary Bucket Recovery

The coordinator initializes the spare bucket as bucket m and sets its level to j_m, determined from the file state. Next, it scans F_2 for 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. This is done through the computation of the LH functions for the keys found in the parity records with $\boldsymbol{g}$ corresponding to m, checking which of these keys initially hashed to m as well. The count of the records recovers the value of r. Next, the coordinator determines every key c of each primary record that was in bucket m at the time of the failure. For each c, it accesses then all the other records that where in the record group. Keys of these records are in the key list of the parity record for c. If any record is unavailable, the recovery is declared unsuccessful. Otherwise, from these records and from the parity record, it recovers each record c in the spare bucket m'. The recovery calculus is usually done as through the XOR of the parity bits with the same offset.

Unsuccessful recovery means a presence of a catastrophic (nonfully automatically recoverable) unavailability of multiple buckets. Such a recovery may still succeed for some records in bucket m [21]. Record groups sharing bucket m are indeed typically spread over different buckets. For at least some records in bucket m, all other members of their groups may thus remain intact, allowing for the recovery.

2.4.3 Recovered Bucket Address Propagation

The new site address s' of bucket m is sent to a client with an IAM. A client unaware of the address change may in the meantime send the query to bucket m to its former address s. Therefore, every query includes in the message its intended bucket number m. 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, it has received the address s' from the coordinator. It got it when it restarted the service, if it failed, or when its bucket m was displaced by the self-detection of the unavailability. In any case, the query is forwarded to address s'. Finally, the client is informed of the displacement of bucket m by an IAM sent by the server s'.

2.4.4 File-State Recovery

The file-state data (n, i) for files F_1 and F_2 have to be recovered into the new primary bucket 0. This recovery precedes that of the records in bucket 0 that requires the file-state data. The algorithm with the proof in [21] starts with the scan of the available buckets of F_1 and of F_2 requesting their addresses m and bucket levels j_m. If there is some m received such that $j_{m-1} + 1$, then one sets n to $n := m$ and i to $i := j_m$. If no such m is found, let M be the largest m retrieved and let i be j_1. If $M + k\, 2^{j1} - 1$, then $n := 0$; otherwise $n := 1$.

2.4.5 Parity Bucket Recovery

Here, the hot spare also becomes the recipient of the recovered parity bucket m. Then, the scan of file F_1 requests every record with record group number ***g*** whose address is m in F_2. The computation uses the split pointer n and file level i of F_2. For every set of primary records with the same g, one computes the parity record and inserts it to new bucket m.

3 Performance

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 is negligible in practice.

The messaging costs in the normal mode for search and scan operations are essentially those of LH.* A rare exception is due to a displaced bucket in which case there is an additional message. An update cost typically doubles, with respect to LH*, to two messages per operation because of the additional message to a parity bucket. Rarely, forwarding in F_2 can cause additional messaging.

*The split cost in normal mode is also only that of LH**, as no access to the parity records is needed. This is a remarkable property, unknown to other high-availability SDDS schemes with record grouping at present. This also comparatively minimizes the file build-up cost.

In the degraded mode, there is additional messaging to and from the coordinator. The computations of various cases are elaborated in [21]. These costs appear typically negligible with respect to those of the recovery.

The first phase of the record recovery is that of the scan of F_2 searching for key c. As there are about M/k buckets in F_2, a successful search thus usually sends $M/k + 1$ messages, assuming the scan sent it by multicast. This is also the cost of an unsuccessful search for c; not counting the message reporting the negative result. If c is found, there are in the second phase between 1 and k, usually k, key search messages for the other primary records in the group. Thus, the typical record recovery costs thus $M/k + k + 1$ messages.

The scan explores buckets of F_2 in parallel. The elapsed time of the record recovery should, therefore, be about that of searching for key c in a *single* parity bucket. The latter is the theoretical minimum for any high-availability scheme. The time difference depends on the implementation and configuration details, e.g., network speed versus CPU speed.

The primary bucket recovery cost is typically $0.7b(2k - 1) + M/k$ messages. Likewise, the parity bucket recovery costs $1 + M$ messages, assuming that each bulk message carries all the records of the corresponding primary bucket. Finally, to recover the file-state costs $M(1 + 1/k)$ messages.

4 Variants

Variations of a generic SDDS scheme enhance selected performance factors at the expense of some others [14]. For LH*g especially, we can enhance the recoverability, at the price of a higher split cost. A variant, called LH*g_1, tolerates multiple bucket failures provided that each failed bucket is in a different bucket group. This is a substantial advantage for a very large file. LH*g_1 avoids the scan of the parity file for record recovery as well. The search for key c in one parity bucket suffices, thus decreasing this operation cost.

In LH*g_1, every record R that moves to new bucket M, appended to the file when bucket n splits, gets a new group key ***g***. The value of g is set to $\mathrm{Int}(M/k)$. The r values are set successively in the order of moves to bucket M, i.e., the *i*th R gets $r = 1$.

For every R, a new parity record ***g*** is created in F_2, including the parity bits. The key of R is removed from the former group of R and the parity bits are adjusted as for the deletion of R. Finally, the counter r of bucket n 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. At the expense of additional messaging to F_2 and of updates to the parity records, the remaining records may, alternatively, simply get new successive values of r, as if they entered an empty bucket.

LH*g usually cannot recover multiple unavailable buckets in different bucket groups. They can contain records of the same record group. In LH*g_1, records with bucket group number g can only be in the buckets forming g. A multiple unavailability of buckets in different bucket groups becomes recoverable. Next, the parity records of each bucket group all may enter a dedicated parity bucket per group at a site different from those for the bucket group. This bucket becomes the only one involved in the recovery of any record within the group. The F_2 scan of LH*g is no longer necessary.

It is possible to generalize LH*g towards parity calculus supporting any n multiple failures within the same group as well. At present, Reed-Salomon erasure correction codes appear the best candidate, [22]. The minimal price is the creation of n parity records per record group.

5 Related Work

We know of many high-availability schemes for centralized environments [23], [7], [25]. Some use mirrors with k^2 times higher storage cost per mirror than for LH*g. Others use the striping, [29], [1]. The striping decreases this cost, but tends to deteriorate the search performance, unlike the record grouping. It also tends to affect the insert performance more. No other scheme using the record grouping is known.

When some existing schemes detect a node failure, they redistribute data over available nodes, [8], [26]. Others replace the unavailable node with a spare node where the unavailable data are reconstructed. The latter strategy seems more efficient for high-availability [25] and to be the best approach when distributed data need to scale [19]. LH*g falls into this class of schemes.

Among the earliest research investigations of schemes for the distributed environment was the RADD (Redundant Arrays of Distributed Disks) scheme, [24], applying the RAID-5 scheme to disks distributed over some sites. Unlike LH*g, the RADD scheme was physical and static, designed for slow networks, and rather inefficient for parallel selections since it used striping.

A scheme based on striping at the physical level exists for distributed file systems with a centralized directory [7]. Likewise, there are mirroring high-availability schemes proposed for distributed or parallel database systems, e.g., Gamma, Tandem, or Teradata machines [6], [26]. Unlike LH*g, they target a single machine or a cluster of a few machines and 1-bucket (site, node) availability.

The known high-availability SDDS schemes were mentioned in the introduction. They offer different storage and access performance trade-offs discussed in depth in [21].

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 are of prime importance. Modern database systems make 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 the expected performance [13]. Experiments with actual applications should follow. For DBMS use, transaction and concurrency management should be addressed. The idea of high-availability through record grouping should also be ported to other SDDS schemes.

Acknowledgments

The authors are grateful for the comments of Jonas Karlsson, Ronny Lindberg, and, especially, of Thomas Schwarz. We also thank the anonymous reviewers whose comments helped the paper to reach its final form.

References

[1] G. Alvarez, W. Burkhard, and F. Cristian, "Tolerating Multiple-Failures in RAID Architecture with Optimal Storage and Uniform Declustering," *Proc. Int'l Symp. Computer Architecture (ISCA-97),* 1997.

[2] W. Bennour et al. "Scalable Distributed Linear Hashing LH$^*{}_{LH}$ Under Windows NT," *Proc. IEEE Fourth World Multiconf. Systems Cybernetics & Informatics and Information Systems Analysis & Synthesis,* 2000.

[3] D. Culler, "NOW: Towards Everyday Supercomputing on a Network of Workstations," EECS technical reprt, UC Berkeley, 1994.

[4] R. Devine, "Design and Implementation of DDH: Distributed Dynamic Hashing," *Proc. Int'l Conf. Foundations of Data Organizations (FODO-93),* Oct. 1993.

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

[6] D. Hsio and D. DeWitt, "Chained Declustering: A New Availability Strategy for Multiprocessor Database Machine," *Proc. Sixth Int'l IEEE Conf. Data Eng.,* 1990.

[7] J. Hartman and J. Ousterhout, "The Zebra Striped Network File System," *ACM Trans. Computer Systems,* vol. 13, no. 3, pp. 275-309, 1995.

[8] S-O. Hvasshovd et al. "A Continuously Available and Highly Scalable Transaction Server," *Proc. Fourth Int'l Workshop High Performance Transaction Systems,* 1991.

[9] J. Karlsson, W. Litwin, and T. Risch, "LH*lh: A Scalable High Performance Data Structure for Switched Multicomputers," *Proc. Int'l Conf. Extending Database Technology (EDBT-96),* Mar. 1996.

[10] D. Knuth, *The Art of Computer Programming,* Vol. 3, Sorting and Searching, second ed. Addison-Wesley, p. 780, 1998.

[11] B. Kroll and P. Widmayer, "Distributing a Search Tree Among a Growing Number of Processors," *Proc. ACM-SIGMOD Int'l Conf. Management of Data,* 1994.

[12] J.-C. Laprie, "Dependable Computing and Fault Tolerance: Concepts and Terminology," *Proc. 15th Int'l Symp. Fault-Tolerant,* pp. 2-11, 1985.

[13] R. Lindberg, "A Java Implementation of a Highly Available Scalable and Distributed Data Structure LH*g," master's thesis, LiTH-IDA-Ex-97/65, Univ. of Linkoping, p. 62, 1997.

[14] W. Litwin, J. Menon, and T. Risch, "Design Issues For Scalable Availability LH* Schemes with Record Grouping," *Proc. Workshop Distributed Data and Structures (DIMACS),* J. Menon, T. Risch, and T. Schwarz, eds., 1999.

[15] W. Litwin, M.-A. Neimat, and D. Schneider, "LH*: Linear Hashing for Distributed Files," *Proc. ACM-SIGMOD Int'l Conf. Management of Data,* 1993.

[16] W. Litwin, M.-A. Neimat, and D. Schneider, "RP*: A Family of Order-Preserving Scalable Distributed Data Structures," *Proc. 20th Int'l Conf. Very Large Data Bases (VLDB),* 1994.

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

[18] W. Litwin and M.-A. Neimat, "*k*-RP$^*{}_N$: A High Performance Multi-Attribute Scalable Distributed Data Structure," *Proc. IEEE Int'l Conf. Parallel and Distributed Information Systems,* 1996.

[19] W. Litwin and M.-A. Neimat, "High-Availability LH* Schemes with Mirroring," *Proc. Int'l Conf. Cooperating Information Systems,* June 1996.

[20] W. Litwin, M.-A. Neimat, G. Levy, S. Ndiaye, and T. Seck, "LH$^*{}_S$: A High-Availability and High-Security Scalable Distributed Data Structure," *IEEE-Research Issues in Data Eng., (RIDE-97),* 1997.

[21] W. Litwin and T. Risch, "LH*g : A High-Availability Scalable Distributed Data Structure by Record Grouping," research report, Univ. of Paris 9 and Univ. of Linkoping, Apr. 1997, http://ceria.dauphine.fr/.

[22] W. Litwin and T. Schwarz, "LH$^*{}_{RS}$: A High-Availability Scalable Distributed Data Structure Using Reed Solomon Codes," *Proc. ACM-SIGMOD-2000 Int'l Conf. Management of Data,* 2000.

[23] D. Patterson, G. Gibson, and R.H. Katz, *A Case for Redundant Arrays of Inexpensive Disks (RAID).* ACM Special Interest Group on Management of Data, 1988.

[24] M. Stonebraker and G. Schloss, "Distributed RAID—A New Multiple Copy Algorithm," *Proc. Sixth Int'l IEEE Conf. Data Eng.,* pp. 430-437, 1990.

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

[26] O. Torbjornsen, "Multi-Site Declustering Strategies for Very High Database Service Availabiity," Thesis Norges Technical Hogskoule, IDT Report 1995.2, 1995.

[27] S. Tung, H. Zha, and T. Kefe, "Concurrent Scalable Distributed Data Structures," *Proc. ISCA Int'l Conf. Parallel and Distributed Computing Systems,* K. Yetongnon and S. Harini, eds., pp. 131-136, Sept. 1996.

[28] R. Vingralek, Y. Breitbart, and G. Weikum, "Distributed File Organization with Scalable Cost/Performance," *Proc. ACM-SIGMOD Int'l Conf. Management of Data,* 1994.

[29] J. Wilkes et al. "The HP AutoRAID Hierarchical Storage System," *ACM Trans. Computer Science,* vol. 14, no. 1, 1996.