LH* Schemes with Scalable Availability

(IBM Almaden Res. Rep. RJ 10121 (91937), (May 1998))

W. Litwin[1], J. Menon[2], T. Risch[3]

Abstract

Modern applications increasingly require scalable, highly available and distributed storage systems. High-availability schemes typically deliver data despite up to $n \geq 1$ simultaneous unavailabilities of the storage nodes (disks, processors with storage, or entire computers), where n is fixed. Such schemes are insufficient for scalable files, since the probability of more than n failures increases arbitrarily with file size. We propose a new schema termed LH*sa withstanding up to n simultaneous unavailabilities with n scaling with the file. We present LH*sa file manipulation and recovery algorithms. We discuss the access and storage performance, and variants tuning selected features. We show that LH*sa files may scale to any number of nodes, keeping the probability of data unavailability arbitrarily small.

1 Introduction

Within a few years, scalability became the key word to modern storage systems, and to the next generation of applications and of computer systems in general, [A&al95], [C&al95], [M97a-c], [FBW97]. Fast processing of large collections of data, required by modern applications, crucially needs *horizontal scalability*, i.e., the ability to run over a fast network of share-nothing sites, or nodes. Of special interest is horizontal scalability over commodity components, e.g., mass produced Wintel or Unix boxes like RS-6000, Alpha etc. and networks like Fast Ethernet or ATM, [A&al95], [I98], [M96], [M97], [M97a]. Large and scalable data collections critically demand high-availability and security of the stored data, [ChS92], [B&al95], [H96], [T95a], [L96], [A&al97]. Traditional high-availability storage techniques provide for limited horizontal scalability, including the hardware RAID schemes, [PGK88], [R97], [KH98], and, more recent software RAID schemes, e.g. within Windows NT and others [SS90], [MRW95], [RM96], as well as the Log-Structured Arrays [M98].

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

[2] IBM Research, Almaden Center, jmenon@almaden.ibm.com

[3] Linkoping University, torri@ida.liu.se

Among various directions, research into horizontal scalability led recently to a new class of data structures termed Scalable Distributed Data Structures (SDDSs). Numerous SDDS schemes have been proposed since 1994. Some are in the References section below, a more complete annotated bibliography is in [SDDS]. The LH* schema is probably best known and most studied [LNS96]. It is based on the well-known Linear Hashing (LH) schema, [L80], [K98], [R98], used in numerous products. Several variants were designed, some with high-availability built-in [LN96a], [L&a97], [LR97]. The SDDSs in general allow data structures to scale-up (horizontally) to very large sizes, i.e., thousands of server nodes and millions of clients[4]. These capabilities open the perspective of TB data storage, and even of PB data collections, supporting billions of transactions [M97b]. LH* in particular, allows a client to find an object (record) identified by its OID (key) in general in two messages, or in four messages at worst, regardless of the number of storage nodes. Combined with the efficient use of large distributed RAM that may now reach, e.g., 8 GB on a workstation, [M97c], SDDSs should lead to performance impossible for more traditional storage systems.

Known high-availability schemes, e.g. variants of the RAID schemes, [PGK88], [BBM93], [BM92], [SS90], [RM96], [W96], and known high-availability variants of LH*, [LN96a], [L&a97], and [LR97], typically guarantee that all data remain available as long as no more than $n \geq 1$ sites (buckets) of the file fail simultaneously. The value of n is a parameter chosen at file creation time. Higher n provides for higher *reliability* that is higher probability that no data is lost in a crash. Such n-availability schemes, (read n as single, double..), suffice for reasonably static files. They may not suffice for scalable files. Whatever is the choice of n, the reliability must decrease when the file grows [H&a94], [NW94]. More flexible schemes are needed with scaling n. We call such schemes *scalable availability* schemes, or *s-availability* schemes in short.

The basic constraint on any s-availability schema is that the change from n-availability to $(n+1)$-availability should be incremental, i.e., without entire file reorganizing. We propose an s-availability

[4] Such numbers could seem futuristic only a few years ago. Today, a popular Web site can get 15M accesses per day easily. In the enterprise world, a popular accounting & management company claims 140K notebooks and WSs accessing its servers daily.

schema called LH*$_{sa}$ ('sa' standing for s-availability) fulfilling this requirement. To the best of our knowledge, it is the first known s-availability schema.

We first define the LH*$_{sa}$ schema with *uncontrolled reliability*. The *n*-availability increases any time the file reaches some size. We present the file manipulation principles and the recovery algorithms. We show that LH*$_{sa}$ files may easily scale up to thousands of nodes and PBytes. We then overview variants tuning some features. Next, we add the capability to control the reliability so it remains close to a desired level. The control reveals necessary in presence of higher failure probabilities. Alternatively, it can improve storage and access performance. Finally, we analyze the related work, and we state the conclusions.

In Section 2 we present the LH*$_{sa}$ scheme. In Section 3, the analysis of its s-availability performance show that the availability of an LH*$_{sa}$ file may remain about independent of the file size. In Section 4 we discuss some design variations. Section 5 discusses the reliability control. Section 6 discusses related work. Section 7 concludes the paper. The Appendix provides details of the scalability analysis in Section 3 and the glossary lists main terms and symbols used.

2 Principles of LH*$_{sa}$

2.1 LH schema*

LH*$_{sa}$ schema is based on the LH* schema, [LNS93], [LNS96]. An LH* file is stored at *LH*-server* computers (nodes), and is used by applications at *LH*-client* nodes. A server is always available for access from clients. A client is autonomous, perhaps mobile, and the initiator of the connections to the servers. The file consists of records, (objects), identified by primary keys, (OIDs), usually noted *c* in what follows. There can be a non-key part of the records, often structured into attributes (fields). A record *R* with key *c* is denoted as *R* or *R* (*c*), or simply record *c* if the non-key part is unimportant in the context. A client can search, insert, or delete a record. Records are stored in *buckets* with a *capacity* of *b* records; $b >> 1$. Buckets are numbered 0,1,2…*M*-1 where *M* denotes the number of buckets in the file. A file has one bucket per server, although different files may share servers. Buckets are basically identified with their servers, unless distinction of roles is necessary. A bucket can be in RAM or on a disk. Searching a bucket in RAM can be orders of magnitude faster than on disk.

The physical (network) address of every bucket is in some *physical allocation table*. These tables are at the clients and the servers. An allocation table can be static, or may dynamically expand with the file. Addresses of new buckets get propagated to clients and among servers as detailed in [LNS96]. Some of the rules will be recalled.

The file is created with $N \geq 1$ buckets, $N = 1$ usually in what follows. The file scales up with inserts, through bucket splits. The splitting and addressing rules of LH* are based on the popular *linear hashing* (LH) algorithm [L80], [K98], [R98]. LH is present in numerous products & platforms: Berkeley-DBM from SleepyCat, Linux, LH-Server for high-performance large client-server applications under Windows-NT or Novell from Revelation Software, jLH-Server in Java, also from Revelation Software, Netscape Browser, Netscape SuiteSpot, Unify-2000 RDBMS… These products are used in turn within dozens of other products.

In LH*, as in LH, every split moves about half of the records in a bucket into a new bucket appended to the file. The splits are done in the order $0,1..N\text{-}1$; $0,1..2N\text{-}1$; $0,1..2^jN\text{-}1,0..$; $j = 0,1..$ The next bucket to split is denoted bucket $\tilde{n}$. The value of $\tilde{n}$ is called the *split pointer*.

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

To perform the splits and the addressing, an LH* file uses a family of hash functions h_l ; $l = 0,1,..$ called *LH-functions*. Each h hashes a key c into bucket address, e.g., $h_l(c) = c \bmod 2^l N$. A split results from the replacement of function h_l currently used for bucket $\tilde{n}$ with function h_{l+1}. This usually re-maps about half of the records into a new address $\tilde{n} + 2^l N$. The coordinator appends the new bucket and moves the records.

Every bucket contains in its header the *bucket level*. This value, usually denoted j, is initialized to $j(m) = 0$ in every bucket $m < N$ when the file is created. The $j(m)$ value indicates that LH-function $h_{j(m)}$ was the last used to split bucket m, or to create bucket m, for $m \geq N$. At any time, and for any bucket m in an LH* file, one only has $j(m) = \hat{\imath}$ or, perhaps, $j(m) = \hat{\imath} + 1$, for some $\hat{\imath} = 0,1,..$ called the *file level*. The coordinator is the only node to store the current values of $\tilde{n}$ and $\hat{\imath}$, collectively called the *file state*. The *correct address*, denoted a, of key c in an LH* file is the address where c should be *dynamically* hashed to, given current values of $\tilde{n}$ and of $\hat{\imath}$. The address a is defined by the LH addressing algorithm [L80]:

(A1) $a \leftarrow h_i(c)$;

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

An LH* client does not access the coordinator for the address computation, to avoid a hot spot. It caches an *image* approximating the file state. The image consists of values noted i' and n' ; $i' = n' = 0$ for a new client, accessing the file for the first time. These values may vary among clients and may differ from the actual $\tilde{n}$ and $\hat{\imath}$. The client uses the image to calculate the address a' = A1 (n', i'). It then sends the request to server a'. The basic requests are: a *key search* requesting record c, an insert, an update, and a delete of record c. The client sends these using unicast (point-to-point) messages.

It might happen that $a' \neq a$. 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. The LH* *test and forwarding algorithm* is as follows, [LNS93]. Address $\hat{a}$ is presumed a:

(A2) $\hat{a} \leftarrow h_j(c)$;

if $\hat{a} = m$ then accept c ;

$\tilde{a} \leftarrow h_{j-1}(c)$;

if $\tilde{a} > m$ and $\tilde{a} < \hat{a}$ then $\hat{a} \leftarrow \tilde{a}$;

forward c to bucket $\hat{a}$;

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

For any SDDS, the correct server that got a forwarded message sends to the client an *Image Adjustment Message* (IAM). For LH*, an IAM contains the j value of server a' to which the client has sent the request. It may also contain its physical address, as well as physical addresses of some buckets preceding bucket a' [LNS96]. The client executes then the *IA-Algorithm*, [LNS93]:

(A3) if $j > i'$ then $i' \leftarrow j - 1$, $n' \leftarrow a' + 1$;

if $n' \geq 2^{i'}$ then $n' = 0$, $i' \leftarrow i' + 1$;

The result of (A3) is a better image, with both i' and n' closer to the actual values. The new image guarantees also that as long as there is no new split, the addressing error that triggered the IAM is not repeated. The physical allocation table at the client may also get refreshed with some new addresses.

(A3) makes LH*-images converge rapidly [LNS93]. Usually, $O(\log M)$ IAMs to a new client (the worst case for image accuracy) suffice to eliminate the forwarding. If a client already has a good image, but

the file starts to scale-up, Algorithm (A3) suffices to keep the incidence of forwarding on the access performance about negligible. In practice, the average key insert cost is one message, and both a successful and unsuccessful key search cost is two messages, regardless of the file size. The worst access performance of an insert or search is four messages, also regardless of the number of nodes of the file. It corresponds to the case of two hops.

These figures translate to access times depending on the network and CPU speeds, and whether the searched buckets are in RAM or at disk [LNS94]. Experiments with the LH* implementation under Windows NT, RAM buckets, and 100 Mbit/s AnyLan network, show the key search time of 200 μs for a 1Kbyte record [B96]. On a Gbit/s network, key search times should be in general under 100 μs, the CPU speed becoming the bottleneck. Similar figures apply to inserts into an LH* file. These times are file size independent, i.e., hold for very large GByte files that can fit into the distributed RAM of multicomputers in larger organizations. They are also orders of magnitude faster than for traditional disk based files and most likely impossible to attain with disk technology. The presence of mechanical parts typically imposes about 10 ms per access at least.

An LH* file also supports *scan search*, or *scan* in short, searching for every record with some non-key values. The client ships a scan in parallel to every bucket. If multicast is available on the network, the scan may be multicast. Otherwise, unicast messages are used. The client's image may not show all the buckets. The algorithm in [LNS96].guarantees that every bucket gets the unicast scan request, and that it gets it only once.

Once a scan is sent, the client should determine when it got the requested replies. A *probabilistic* termination, or *probabilistic scan*, consists of setting a time-out for the reception of every new record. Only servers with the selected records reply. Alternatively, the client sets up a *deterministic scan*. It checks whether all the selected record have arrived. One protocol for a deterministic termination [LNS96] is that every bucket replies with at least its address m, its level j_m, and the selected records, if any. The client terminates when it has received, in any order, $m = 0$ and $m = 1$ etc., up to $m = 2^i + n$, where $i = \min(j_m)$, and $n = \min(m)$ with $j_m = i$. This guarantees that all the buckets replied.

The principles of LH* lead to many variants with performance tradeoffs [LNS96], [KLR96]. There are several ways to perform load control, increasing the average load factor over the basic value of 70 %. There

are also many ways to perform a split and to organize the bucket interior. One may also design LH* schemes without a coordinator [LNS93a].

2.2 The *LH*$^*_{sa}$ file structure*

An LH*$_{sa}$ file F consists of an LH* file F_0 called the *data* file and of $i = 1..J$ *parity files* F_i. The file-state data of F consist of values ($\tilde{n}$, $\hat{\imath}$) of F_0 mirrored at all buckets 0. Records of F_0 are called *data* records (objects), Fig. 1, or simply records. A data record c is the user record c sent for storage. Each parity file contains *parity* buckets 0,1.. with *parity* records for the high-availability. Parity records contain check information used to achieve the desired level of availability. An LH*$_{sa}$ file consists of at least (F_0, F_1) with files $F_{i>1}$ progressively added when F_0 scales. Every F_0-bucket m contains in its header in addition to j_m a value called *bucket availability level* usually denoted i or i_m below. This value denotes the number of parity files F_i associated with the bucket. As it will appear, i_m also indicates the number of parity buckets, and of parity records per data record in bucket m. All data and parity buckets are basically at different servers.

Every LH*$_{sa}$ file is provided with a family of *grouping functions*, noted f_i ; $i = 1,2..$ Each f_i provides to every data bucket a number g_i, called *bucket group number*. A *bucket (availability) group* consists of all buckets sharing g_i. Functions f_i are chosen so that all bucket groups are of the same size in the number of participating buckets, denoted k. Every data or parity record has furthermore some distinct *rank* within its bucket noted r ; $r = 1,2..$ This is basically the position of the record in the bucket. Each data record c is provided with i_m *record (availability) group numbers* that are pairs (g_i, r). A *record (availability) group* consists of all data records sharing the same (g_i, r). A record group size is at most k, as it will appear. Every record group (g_i, r) is provided with a parity record with rank r stored in bucket g_i of F_i. Hence, all the parity records of record groups within the same bucket group g_i are in the same parity bucket g_i. The value (g_i, r) is considered the key of the parity record. Fig. 1 shows the structure of a parity record. It contains the keys c of all the records in group (g_i, r), and the parity bits B. The bits suffice to recover the non-key data of every single unavailable group member. Example 2.3.2 below illustrates this capability more in depth.

Parity records are created or updated when the application makes an insert or an update of a record c. The correct data bucket m stores or updates record c. For every parity file F_i ; $i = 1..i_m$; it also forwards the new record c with its rank r to bucket g_i. If record c was only updated, it sends to g_i the update record where

only the bits that have changed are set to 1. Bucket g_i either has, or has not yet, the parity record (g_i, r). Accordingly, record (g_i, r), with rank r, is created or updated.

Fig. 1 Record structure in LH*$_{sa}$ files

Each value g_i for a record c is calculated from its correct data bucket address m. The grouping functions $f_i : m \rightarrow g_i$ are:

$g_1 = \text{int}\,(m / k)$

$g_2 = \text{mod}\,(m / k) + \text{int}\,(m / k^2)$

$g_3 = \text{mod}\,(m / k^2) + \text{int}\,(m / k^3)$

....

$g_i = \text{mod}\,(m / k^{i-1}) + \text{int}\,(m / k^i)$.

The groups generated by each function are:

f_1 : $(m, m+1, m+2 .. m+(k-1))$ for $m = 0, k..$

f_2 : $(m, m+k, m+2k .. m+(k-1)k)$ for $m = 0,1,2..k-1, k^2, k^2+1..k^2+(k-1), 2*k^2..$

f_3 : $(m, m+k^2, m+2k^2 .. m+(k-1)k^2)$ for $m = 0,1.. k^2-1, k^3, k^3+1.. k^3+(k^2-1), 2k^3..2k^3+(k^2-1)$

...

f_i : $(m, m+k^{i-1}, m+2k^{i-1} .. m+(k-1)k^{i-1})$ *for* $m = 0,1..k^{i-1}-1, k^i .. k^i + k^{i-1} - 1, 2k^i ...$

Example

Consider $k = 4$. The following groups are generated and illustrated in Fig. 2. Groups of f_1 and of f_2 are shown arranged so their numbers appear as usual 2-d coordinates. Groups of f_1 appear horizontal and those of f_2 are vertical. Bucket addresses correspond to points, e.g. bucket 9 is in groups $g_1 = 2$ and $g_2 = 1$. Members of f_i -groups with $i > 2$ are along the diagonal lines. The members of groups f_1, f_2, f_3 and of f_4 involving element 0 are respectively shown in italics, **bold**, ***bold italics***, and, finally, underlined.

f_1 : 0 = (0,1,2,3), 1 = (4,5,6,7), 2 = (8,9,10,11)..

f_2 : 0 = (0,4,8,12), 1 = (1,5, 9,13).. 3 =(3,7,11,15), 4 = (16,20,24,28)..7 = (19,23,27,31), 8 = (32,36..44)..

f_3 : 0 = (0,16,32,48), 1 = (1,17,33,49).. 15 = (15,31,47,63), 16 = (64,80, 96, 112)..

...

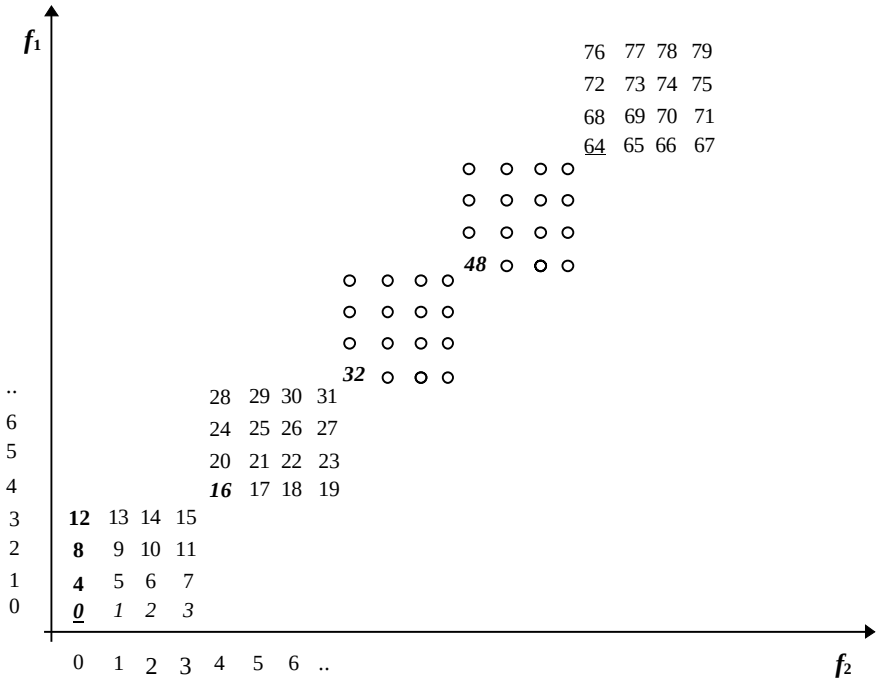


Fig. 2 LH*$_{sa}$ groups for k =4 and J = 4.

The following easy to see properties hold for the LH*$_{sa}$ grouping functions:

- Every f_i partitions the set of numbers {0,1,2..}.
- Every data record in a record group (g,r) is in a different bucket.
- For every bucket address m_1, m_2 and grouping function f_i, if m_1 and m_2 belong to the same bucket group g_i generated by f_i ; i = 1,2.. ; then for every f_j ; $j > i$; m_1 and m_2 belong to different groups g_j.

These properties allow for the scalable availability when employed as follows. The basic schema for LH*$_{sa}$ we first address is called LH*$_{sa}$ with *uncontrolled reliability*. The meaning of this qualifier is explained in Section 5.

2.3 File expansion

2.3.1 Overview

An LH*$_{sa}$ file is created with the f_1-groups, basically with one (N = 1) bucket, and any convenient k value, provided that $k = 2^l$ for some $l > 0$. The value i_0 in bucket 0 is initialized to 1. For every insert, a parity record in F_1 is created or updated. The f_1-structure provides trivially 1-availability, allowing recovery from any single bucket unavailability. When the file scales up, the f_2 record groups start to be added. Their creation starts with the split creating bucket k. This is the split of bucket 0, given that $k = 2^l$. The f_2 parity records are from now on created for every record at each split. The i_m value in the split bucket m is set to i_m :=: $i_m + 1$. The i value in the new bucket created by the split is set to the new i_m as well. This process creates f_1 and f_2-groups, for all the buckets up to bucket $k^2 - 1$.

If the file scales up to bucket k^2, the f_3 - groups start. Bucket k^2 is also created by a split of bucket 0. From now on, every split of a bucket gracefully builds-up the f_3 - group structure for the records in the bucket and those moving out. It also updates to $i = 3$ the header of each split bucket, and of each new one. When F_0 reaches bucket $k^3 - 1$, the whole file bears f_3 - groups. Once a split of bucket 0 creates bucket k^3, the grouping according to f_4 starts. The process continues with grouping using f_4 starting when bucket k^4 is created etc.

The value i_m at each data bucket m is the number of bucket groups g_i bucket m currently participates in. It also indicates the buckets g_i that should be updated by an insert or an update to bucket m. The value I = min (i_m) ; m = 1..M – 1 ; is called *file availability level*. As it will appear soon, an LH*$_{sa}$ file is a I-availability file. At any time, a bucket with LH-level $\hat{\imath} + 1$ has the grouping level $i = J$ where $J = I + 1$ or $J = I$. In turn, the bucket not yet split, still using $h_{\hat{\imath}}$, carries the grouping level $i = I = J$ or $i = I = J - 1$. Hence, at any time, there are at most two grouping levels in F_0.

2.3.2 Example

Fig. 3 shows an evolution of an LH*$_{sa}$ file. The file is created with data and parity buckets 0. Their capacities are assumed b = 4, and bucket group size is k = 2. Record structures are in Fig. 1. The keys of data record are italic. Only the first two bits of the non-key part of a data record or of the parity bits are materialized in the figure. Even parity is assumed, hence the total number of bits equal to 1 at the same position within the record group should be even. As usual, each parity bit allows therefore for the recovery

of a single missing bit at the same position within an unavailable data record. Also as usual, updates to parity records are performed using the XOR operation.

Fig. 3a illustrates the file state after initial five inserts. Four data records are stored in data bucket 0. The record ranks r = 1..4 correspond to the positions, e.g. record 21 has rank 3. Bucket 0 also resends each insert with its rank to bucket $g_1 = 0$ of F_1, to create or update the parity record with the same rank. The f_1-groups correspond to the horizontal groups in Fig. 2. Every record group contains one record. Hence parity records are initially replicas of the data records.

The fifth insert creates an overflow of bucket 0. The bucket stores record 3 as an overflow record. It also resends it to F_1, together with its record group number (0, 5). It reports the overflow situation to the coordinator that requests bucket 0 to split. Since $j_0 = 0$, bucket 0 uses LH-function h_1 to split. The example considers the family of LH-functions $h_i : c \rightarrow c \bmod 2^i$. The split creates bucket 1 of F_0 and moves there all records with odd keys. The result is in Fig. 3b. In both buckets, j_m is set to 1 after the split. Note that the overflow was resolved, i.e., record 3 is stored within the bucket.

The data records remaining in the split bucket may get new ranks, e.g., record 30 in data bucket 0 at Fig. 3a. Records that move get successive rank in the new bucket which are here r_1 ranks in bucket 1. Fig. 3b shows the new situation in our example. Both data buckets 0 and 1 send messages about the new ranks to the parity bucket 0 in F_1. This bucket updates the parity records accordingly, as in Fig. 3b. The result is three parity records. This number is the highest r after the split in both buckets 0 and 1 at Fig. 3b. Note that this is about half less than before the split, and that the overflow disappeared in consequence. The parity records (0,1) and (0,2) provide now each the parity for k = 2 data records, with keys 12 and 21, and 30 and 15, respectively. The size of these record groups is thus that of the bucket group. The parity bits are recomputed and typically change, e.g., bits in record (0, 1) become 01. Record (0,3) corresponds to only one data record. The size of the corresponding record group is thus less than that of a bucket group.

Fig. 3c shows the situation after three more inserts. The insert of record 42 added a record to record availability group (0,3) and modified its parity record (0,3). Inserts of records 7 and 13 started record groups (0,4) and (0,5) and led to the corresponding parity records. The overflow situation generated a message from bucket 1 to the coordinator. According to LH* principles it triggered the split of bucket 0 of F_0, using h_2. Record 13 remained temporarily stored as an overflow record at bucket 1. Split of bucket 0

generated bucket 2 of F_0, as illustrated in Fig. 3d. It led to the update of the parity records in bucket 0 of F_1 corresponding to data records remaining in bucket 0. Records that moved to bucket 2 also got new bucket group number g_1, $g_1 = 1$. This lead to the creation of bucket 1 of F_1 and of the parity records (1,0) and (1,1).

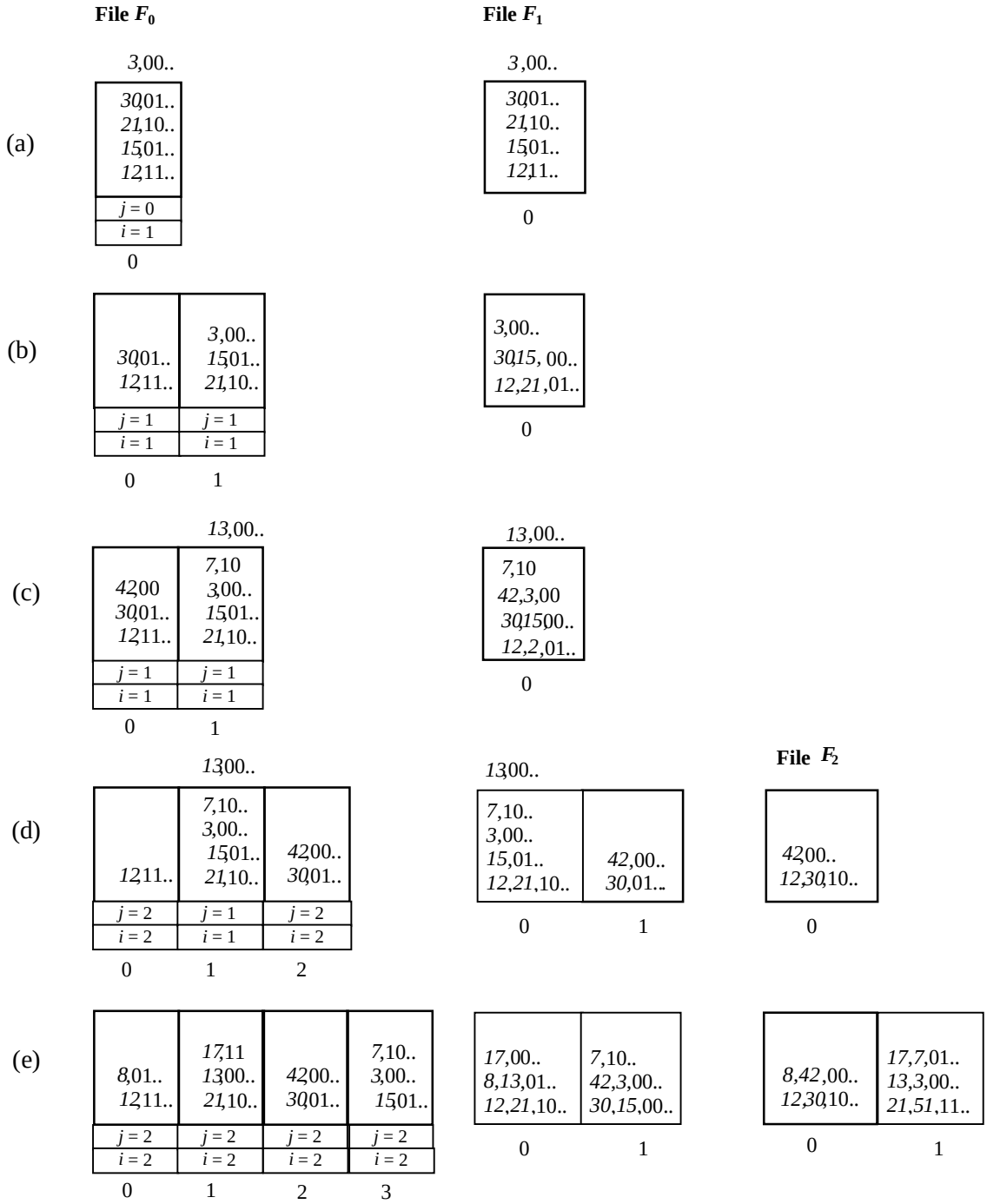


Fig. 3 Evolution of LH*$_{sa}$ file

Since $k = 2$, the split also led to the creation of the F_2 file. Accordingly, i was set to 2 in data buckets 0 and 2. Two parity records according to f_2 were also created in F_2 bucket 0. They group same rank records in bucket 0 and/or bucket 2 of F_0. The f_2 - grouping corresponds to vertical groups in Fig. 2. Records in bucket 1 of F_1 remain grouped according to f_1 only.

Fig. 3e shows the file after two more inserts, of records 8 and 17. The latter insert triggered the split of bucket 1, using h_2 again. The inserts, and the changes to values of r due to the split, triggered updates to some parity records in both F_1 and F_2. Note that the split suppressed the overflow in F_1 bucket 0 and decreased the total number of F_1 records. It also created parity records in F_2 for every data record that was in data bucket 1. At this point, every data record is grouped according to f_1, and to f_2 and has two parity records. New parity records in F_2 led to creation of its bucket 1.

Since F_0 reaches $k^2 = 4$ buckets, the split pointer $\tilde{n}$ points again to bucket 0 that is next to split, using h_3. This split will start the f_3 grouping, creating bucket 0 of F_3. Further splits will progressively create f_3-groupings for all other buckets of F_0. If the inserts continue the expansion will continue with h_4 and f_4 etc., to any size needed.

Note that in Fig. 3 the indexes j of LH-functions and i of grouping functions have the same values. This property is however valid only for $k = 2$.

2.4 High-availability

2.4.1 Overview

In general, file F is said to be *available* if all the user data stored in F are available for access. File F is a *high-availability* file if it remains available even if some of data records or buckets are unavailable. An n-availability file, (read n as single, double, triple...), remains available despite the unavailability of any n buckets, while the unavailability of some $(n + 1)$ buckets compromises its availability. The unavailability of $m > n$ may be still recoverable for some buckets, but not for all. These recovery capabilities are schema specific. Finally, F is a *scalable-availability* (s-availability) file if n grows (or shrinks) with the file. Other facets of availability management, e.g., concerning the management or recovery of the code, such as the coordinator, running at servers and clients, are beyond the scope of the LH*$_{sa}$ design. See [T95a] for instance for a discussion of the related issues.

The example illustrates the high-availability features of the LH*$_{sa}$ schema. Grouping by f_2 allows every pair of records c_1, c_1' in the same f_1 record group (g_1, r), to become members of different groups (g_2, r) and (g_2', r). There is no other record of (g_1, r) in (g_2, r) or ($g_2^{'}$, r). If c_1 and c_1' both fail, they cannot be recovered using parity bits of (g_1, r). However, c_1 can be possibly recovered using parity bits of (g_2, r). Likewise, c_1' can be recovered from (g_2', r). Adding f_2, thus allows for 2-availability.

For instance, in Fig. 3e record 12 in bucket 0 is f_1 - grouped with record 21. If bucket 0 fails, one can recover record 12 from record (0, 1) in F_1. If record 21 is unavailable as well, the record (0, 1) in F_1 does not suffice. Record 12 can then be recovered from record (0, 1) in F_2. Likewise, record 21 can be recovered from record (1, 1) in F_2. In fact, it can be alternatively recovered from record (0, 1) of F_1, after the recovery of record 12.

In contrast, f_2 does not allow for 3-availability. If c_1, and the parity records of its record groups (g_1,r) and (g_2, r) fail, c_1 is not recoverable. This would be for instance the case of the failure of the records: 12 in F_0, (0, 1) in F_1, and (0, 1) in F_2. More generally, in Fig. 2 it would be the case of unavailability of a record in any data bucket, together with the parity records of both its vertical and horizontal groups. In such cases, the unavailable data record can be recovered using its f_3 group. The necessary f_3 bucket should be available, assuming that at most 3 records can fail simultaneously. Creating f_3 groups, allows thus for 3-availability. By the same token, adding f_4 allows for 4-availability etc. The *n*-availability may scale up in this way to any degree required by the file size.

The notable overall result is that the increase to the high-availability in LH*$_{sa}$ file is done incrementally, one existing bucket at the time. No global reorganizing occurs. The behavior of LH*$_{sa}$ files has the goal highlighted in the Introduction as of primary importance for s-availability files.

The next sections present unavailability detection and recovery in an LH*$_{sa}$ file in depth. We first discuss unavailability detection and the overall principles of recovery. Next, we present the recovery algorithms. We start with the basic algorithms recovering respectively a data bucket and a parity bucket in presence of a single unavailability. We then show the full LH*$_{sa}$ bucket recovery algorithm, combining the basic ones to recover from a multiple unavailability. Afterwards, we show the record recovery algorithms, reconstructing a single record subject to the key search in an unavailable bucket. If found, the record is not restored in the file, but only provided to the application, avoiding the need to wait for the full bucket

recovery. Finally, we address file state data recovery and the self-detected recovery of a bucket. For each algorithm, we prove its correctness that it provides 1-availability or *I*-availability, respectively.

2.4.2 Unavailability detection and overall recovery management

A client or a server attempting to access some bucket *m*, at some physical address *s*, can detect its unavailability. The coordinator requesting a split of bucket *m* can also detect it. Finally, bucket *m* restarting from an unavailability during which no access to it was requested, can self-detect its unavailability as well.

The client or the server notifies the coordinator. The failed bucket may be a data bucket, and/or a parity bucket. The recovery of data bucket 0 should include the file state data $(\tilde{n}, \tilde{i})$ of all the current files F_i. Specific recovery actions are also required when the unavailability is self-detected.

Bucket *m* is recovered at some spare server at address $s' \neq s$. It is assumed that a spare server is always available. 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. The record recovery should typically complete the search faster than the full bucket recovery. It either delivers the record or determines that there is no such record in the file.

The location change of bucket *m* to address *s'* is sent out to clients and servers using IAMs. 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*. In this case, it sends the request to address *s* which is then forwarded to the correct address, as detailed below in Section 2.5.

2.4.3 Basic data bucket recovery

The coordinator needs to recover every record in the (single) unavailable data bucket *m*. Algorithm (A4) does it for a single unavailability. Any parity file could be used, we assume F_1 below since we deal with 1-availability. For didactic reasons, we describe (A4) as if the coordinator directly used it. However, the coordinator actually calls Algorithm (A6) presented later for recovering from a multiple unavailability, which in turn calls (A4) internally. If it happens that $m = 0$; the coordinator first recovers the file state, as it is shown in Section 2.4.8. In Step 1 of (A4) the coordinator initializes the spare bucket. Step 2 finds in F_1 every parity record with keys of data records that were in bucket *m*. In Step 3, all F_1 buckets with such

records reconstruct the missing records and send them to the spare. In Step 4, the spare inserts all these records and recovers the r_m value. Steps 2 - 4 are executed basically in parallel.

Algorithm (A4) Data bucket recovery (case of $n = 1$)

Let bucket m be the (only) failed one, and let g be its bucket group number according to f_1.

1. The coordinator creates the spare bucket m', and initializes its header with the values of j and of i that were in the unavailable bucket.
2. The coordinator issues a query, let it be Q_1, to bucket g in F_1. Q_1 requests the following manipulation.
3. For every record (g, r) containing data key c that is LH-mapped to bucket m given $\tilde{n}$, and $\hat{\imath}$:
   - 3.1 If the record has only one data key, then bucket g reconstructs the data record from the parity bits, and sends it to bucket m', together with r.
   - 3.2 Otherwise, bucket g searches in F_0 for every record c' with key $c' \neq c$ in record (g, r), then reconstructs record c from these records and from record (g, r). Finally, it sends record c to bucket m', together with r.
4. Bucket m' inserts record c with rank r.

Proof. One has to prove that under the single bucket unavailability assumption Algorithm A4 reconstructs all and only the unavailable data records, and with the original ranks. Under the single bucket unavailability assumption, parity bucket g is available and all the data buckets with bucket group number g except for bucket m are available. The file state being available, or recovered earlier, for $m = 0$ the coordinator can always initialize the spare bucket with j and i. Every record c in bucket m had to be LH-mapped to address m, given the F_0 state. Every related parity record has to contain key c and no other parity record can contain c. Step 3 finds all these parity records through Q_1. A record (g, r) found may have one or more keys in it. First case corresponds to key c alone in record (g, r), and to absence of other data records in record group (g, r). The content of record (g, r) suffices then alone to recover the unavailable record c; this motivates Step 3.1. Otherwise the data keys in record (g, r) correspond to all and only the data records necessary and sufficient with record (g, r) to reconstruct record c. Step 3.2 is always able to bring all these records to bucket m' with record (g, r). All the corresponding data buckets must be indeed available (under the single unavailability assumption). It therefore can recover every corresponding unavailable record.

Step 3 reconstructs all the unavailable records, since it loops over all and only the corresponding keys. Finally, Step 4 puts all the records within the spare with their original ranks. @

2.4.4 Basic parity bucket recovery

This is done through Algorithm (A5). As for (A4), for didactic reasons, we describe (A5) as if the coordinator directly used it. However, it is actually also called through Algorithm (A6) that basically reduces to (A5) if only one parity bucket is unavailable. Algorithm A5 works for every F_i.

Algorithm (A5) Parity bucket recovery (case of $n = 1$)

1. Let bucket g_i be the failed parity bucket. The coordinator initializes a hot spare as new recipient of bucket g_i. It also instructs it to send query Q_2 to all buckets m of F_0 whose bucket group number is g_i. Q_2 requests every record within these buckets with its rank r.

2. For every set of records with the same (g_i, r), the spare reconstructs the parity record and provides it with rank r.

Proof. Under the assumption of single unavailability, all the buckets to receive Q_2 are available. The bucket address m of every bucket with bucket group number g_i can be easily computed from formulae for f_i in Section 2.2. Since Q_2 requests also rank r of each record, it brings all the data records required to recover every parity record, and only those records. @

Actual implementation of Q_2 lead to design choices we address in Section 4.

2.4.5 LH*$_{sa}$ bucket recovery

Algorithms (A4) - (A5) provide for 1-availability. Multiple unavailability may concern data buckets, and/or parity buckets, and/or file state data. Algorithm (A6) below combines (A4) and (A5) to provide I-availability for a file with the availability level I, i.e. with at most $J = I + 1$ parity files. (A6) assumes that file state data is available when it is launched.

(A6) is initially called by the coordinator when unavailability of bucket m is detected. This can be a data bucket or a parity bucket in F_1. Accordingly, (A6) starts the recovery using (A4) or (A5) with F_0 and F_1. This attempt may succeed or fail. In the latter case, bucket m initialized at the spare by each algorithm is deleted. The reason for the failure may be further unavailability of data or parity buckets. In the latter case, if $I = 1$, or there are more than I failed buckets, the recovery ends up unsuccessfully. Otherwise, the recovery restarts using F_0 and F_2. This phase concerns bucket m if it was a data bucket, or one of the data buckets whose unavailability was discovered by (A5). This recovery succeeds or fails again, in which case one continues using F_3 provided that $I > 2$ etc., up to F_J. If one succeeds, the bucket recovery continues for all other data buckets discovered unavailable. If all these buckets are recovered, then the recovery of the

parity buckets starts. These may uncover further unavailable data buckets. If so, the recovery switches back to those. If it works out, then (A6) returns to the recovery of the remaining parity buckets. If all this succeeds, (A6) terminates successfully.

This principle leads to two lists in (A6). List B_1 contains the addresses of the unavailable data buckets. List B_2 contains the addresses of the unavailable parity buckets, including the corresponding i values. When (A6) is initially called, one list has 1 element and the other list is empty. Boolean C indicates the success of bucket recovery by (A4) or (A5). (A6) succeeds iff it returns both lists empty. During the execution of (A6), the lists are maintained by A4 and A5. These are assumed modified adequately with respect to their basic description. If (A4) or (A5) succeeds with the recovery, it removes the bucket from the list. The algorithms also set C according to the success or failure of each recovery.

Algorithm (A6) LH*$_{sa}$ bucket recovery

```
While  B1 ≠ ∅
        For each m ∈ B1
        C =: .false
            while C =: .false call  A4 (m, i)
                    if C =: .false then
                            if  i = I  or (|B1| + |B2|) > I) then exit  end if
                i = i + 1
        end while
        i =: 1
   End for
   if B2 = ∅  then exit
   else  if  (|B2|) > I) then exit end if
   For each m ∈ B2
        call A5 (m, i)
   End for
   if B1 = ∅  and B2 = ∅ then exit end if
   if  (|B1| + |B2|) > I) then set C = .false ; exit  end if
   else i =: 1
end  while
```

Proof. Consider that the coordinator calls (A6) because of unavailability of data bucket m. (A4) may succeed, leading to empty B_1 and empty B_2. This will make the recovery successful, and, accordingly (A6)

will exit. Alternatively, (A4) may detect the unavailability of the parity bucket. In this case, *m* remains in B_1, and (A4) updates accordingly list B_2. Bucket *m* cannot be recovered from F_1. If $I = 1$, the recovery is impossible and (A6) terminates. (A6) also terminates unsuccessfully if the number of all unavailable buckets exceeds *I*, as the LH*$_{sa}$ scheme is designed to provide *I*-availability only[5]. If neither of the conditions occurs, the recovery of bucket *m* may succeed if F_2 is used. (A6) continues accordingly, calling (A4) for $i = 2$. It loops in this way until either (i) the bucket recovery succeeds, or (ii) unsuccessful termination conditions are met on *i* or (iii) the number of unavailable buckets, triggering the failure of (A6).

If recovery of bucket *m* is successful, (A6) continues in the same way with every other element in B_1 being an unavailable data bucket discovered on the way. As a result, either all these buckets are recovered or (A6) exits unsuccessfully. If they are all recovered, B_1 is empty, and the first **for** loop terminates. (A6) then starts the recovery of all the unavailable parity buckets. In particular, the first **for** loop is skipped entirely if the initial bucket *m* is a parity bucket.

The 2nd **for** loop recovers the remaining parity buckets that are all in B_2. It may succeed for all or only some of them. In the former case, the recovery should terminate successfully. Since both lists become empty, (A6) will exit accordingly. In the latter case, some unavailable data bucket had to be encountered. Hence, B_1 is no longer empty, and these data buckets should be recovered, before the remaining parity buckets in B_2, are recovered in turn (hopefully). The recovery of the data buckets should start from F_1, being carried out as already described. That is why (A6) resets *i* to 1. Since the overall **while** loop tests the emptiness of B_1, (A6) will come back to the recovery of data buckets, as it should.

If there is at most *I* unavailable buckets, (A6) always terminates successfully. If F_J is reached and explored unsuccessfully, which causes (A6) to exit unsuccessfully, it means that more than *I* unavailable buckets are in *F*. The same occurs when there are more than *I* elements in both lists. Hence (A6) provides the *I* – availability, as intended.

(A6) should always terminate in practice. There is however no theoretical guarantee on its termination. If buckets fail often enough, it is easy to see that (A6) can indefinitely race through its **for** loops.

---

[5] It is possible to modify (A6) so that one recovers sometimes more than *I* failed buckets, as shown in Section 4.1

2.4.6 Basic record recovery

The *record recovery* Algorithm (A7) reconstructs a data record c subject to a key search in an unavailable data bucket m, or determines that record c was not in the file. It assumes that bucket m is the only one unavailable. It can be performed, using any parity file F_i. The recovered record is not restored in the file. The purpose of (A7) is only to speed-up the search, through concurrent execution with bucket recovery. The latter may be longer than record recovery, since it has to recover $b >> 1$ records and to create a new bucket.

Algorithm (A7) actually is only a part of Algorithm (A8) dealing with multiple unavailability. Step 1 retrieves the parity record from F_i. In Step 2, if no parity record is found, 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 several records in the group.

Algorithm (A7) Record recovery (case of $n = 1$)

1. The coordinator computes the group number g_i of bucket m and sends query Q_3 to parity bucket g_i requesting parity record (g_i, r) containing c.

2. If Q_3 terminates unsuccessfully, the search for c terminates as an unsuccessful key search.

3. If c is the only key in record (g_i, r) then record c is reconstructed from the parity bits in record (g_i, r) only.

4. Otherwise, for every key $c' \neq c$ in record (g_i, r), the coordinator issues to F_0 a key search for record c'. If all records c' are received, then, record c is reconstructed from these records and the parity bits.

Proof. One has to prove that (A7) provides the searched record c if it were in the unavailable bucket m or determines for sure that it was not there. Under the single unavailability assumption, bucket m is the only one unavailable in F. Record g is in F_i iff record c was in bucket m. Hence, Step 1 always terminates and determines whether record c was in F_0. If so, Q_3 brings the parity bits for record group g and keys of all its other members. If record c was the only one in the group, the parity bits in record g suffice for its reconstruction, in Step 3. Otherwise, Step 4 always brings all other members, since they are all in buckets other than bucket m, and all these buckets are available. These records and parity bits in record g suffice to reconstruct record c. @

2.4.7 LH*$_{sa}$ record recovery

Algorithm (A8) below provides record recovery in the presence of up to I unavailabilities for a file with J parity files F_i. It uses (A7). The coordinator calls (A8) starting with (A7) on F_1. If this attempt does not find record c, and it encounters a failure of a bucket in F_1 or in F_0, then the recovery ends as unsuccessful if $I = 1$. Otherwise, it continues using F_2, etc., until F_J is explored. If it still encounters failures prohibiting the completion, it exits unsuccessfully.

(A7) in (A8) is assumed slightly modified from the basic (A7) described previously. It sets a Boolean C according to the result of the recovery attempt for each F_i. The C value also indicates the overall completion of (A8) to the coordinator. As for (A5), the successful completion of (A7) within (A8) either returns record c or the indicator that the record was not in F_0. In addition, it returns lists B_1 and B_2 for the coordinator, to launch the corresponding bucket recovery.

Algorithm (A8) LH*$_{sa}$ record recovery

$i = 0$; C =: .false
while C = false and $i < I$ do
  i =: $i + 1$
call A7 (m, i)
end while
exit

Proof. If there is a single unavailability, (A8) reduces to (A5) applied to F_1. Hence (A8) succeeds. If there are J files F_i and up to J unavailabilities, then in at least one of F_i the parity bucket with c and all the data buckets in the group of bucket m must be available. Hence, (A8) will succeed. Finally, if F_I is reached and an unavailability prohibiting recovery of record c is still encountered, there were at least $I + 1$ unavailable buckets in F. Hence, (A8) may exit unsuccessfully.

2.4.8 File-state recovery

The file state data ($\tilde{n}$, $\hat{\imath}$) are necessary for Algorithm (A4), and (A5). Hence, if bucket 0 is recovered, they have to be recovered first into the new data bucket 0. These data are mirrored at all buckets 0, as they are negligible in size and updated very infrequently. Since there are J +1 buckets 0, I – availability of state data is trivially achieved.

2.4.9 Self-detected recovery

Bucket *m* can self-detect unavailability through a local test determining that its data are corrupted to the point it cannot recover locally. It can also detect that it was restarted with the correct data from a temporary unavailability. In both cases, bucket *m* contacts the coordinator before serving any file requests. In the first case, it requests 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 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 provides it with the address of the new bucket *m*. The hot spare *m* needs the address of its replacement to update its physical node allocation table.

2.5 File manipulation

An application manipulates an LH*$_{sa}$ file as an LH* file. Internally, each manipulation is enhanced for unavailability management. The general rule is that manipulations encountering unavailability are passed to the coordinator for completion. In case of an insert, the client site serving the application 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 asynchronously performs the appropriate actions and completes the actual insert of record *c* into the correct bucket.

As it was outlined in Section 2.4.2, it can also happen that an LH*$_{sa}$ client sends a key search or insert to a former server of a bucket that was displaced. To resolve the addressing given this constraint, the client always includes in the message the intended bucket number, let it be *m*. This principle applies to clients of F_0 and to servers of F_0 acting as clients of any F_i.

Let it be server *s* to which the client sends the message. If it is unavailable, the client resends the message to the coordinator, as was discussed in Section 2.4.2. 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 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. In both case (ii) and (iii), an IAM is sent to the client with the new address of bucket *m*.

A query can also encounter the unavailability of a forwarding bucket. The sender of any such query, a client or a server, re-sends the query to the coordinator. The coordinator sends any such query to its correct bucket, bypassing the forwarding through the use of the file-state data and of Algorithm (A1). If the correct bucket appears unavailable, it initiates the record or bucket recovery.

Under these rules, an LH*$_{sa}$ key search for a record in an available bucket does not generate access to any F_i. If each forwarding bucket is also available, and the correct bucket is not displaced, the search performs as in the LH* file. An insert adds r to the record to produce the data record as described in Section 2.3. It also sends the data record to $F_1..F_i$ according to the i value found in F_0 bucket header. Scans translate to parallel searches in F_0. If all F_0 buckets are available, a deterministic scan works as in the LH* file. Otherwise, the coordinator is alerted, as discussed above. If a probabilistic termination is requested, the scan basically works as in the LH* file. It basically cannot detect unavailabilities. Some detection can still come from the network level, but such implementation specific capabilities are beyond the LH*$_{sa}$ design.

A request for deletion of record c from an LH*$_{sa}$ file causes the deletion of data record c. The corresponding parity records are updated. A parity record is deleted, if no data key remains in it. We do not elaborate on deletions further in this paper.

3 Scalability analysis

3.1 High-availability

We recall from Section 2.4 that one measure of high-availability is the n-availability. This is a deterministic measure that depends solely on the file schema. Depending further on the schema, the unavailability of $m > n$ buckets may be recoverable for some, but not all buckets. An LH*$_{sa}$ file is always I-available. An unavailability of $m > I$ buckets may be recovered for some buckets, provided enhancements to the algorithms above. These issues are discussed in Section 4.1.

A related probabilistic measure, among popular metrics, [BM92], [M94], [M98], [KH98], is the *reliability*. It is the probability P that the (entire) file is available, [BM92]. Higher reliability means lower probability of data unavailability, i.e., of an unavailability of $m > I$ buckets. One typically expects P to remain above some reasonable threshold T, e.g. $T = 0.9$. P depends on n, on probability p that a bucket fails, and on number of buckets M in the file. Two important well-known properties link the availability level and the reliability [H&a94]:

1. For every given M, P monotonously increases towards 1 with n (but never reaches 1, obviously). In other words, increasing the availability increases the reliability.
2. In contrast, for every given n, P decreases arbitrarily close to 0 when M grows, i.e. when the file scales up. In other words, increasing any n-available file decreases its reliability.

These properties will be discussed for LH*$_{sa}$ schemes. It will appear that scaling the availability according to LH*$_{sa}$ schema, may effectively counter-balance Property 2.

In an LH*$_{sa}$ file with J parity files, either every bucket has the grouping level i equal to $i = J = I$ or some have $i = J - 1$. In the former case, there are I parity records per every data record. To estimate P of an LH*$_{sa}$ file, one may start with that case. We denote the corresponding file as LH$_{sa}^{I}$ file and we denote P as P_I. An LH$_{sa}^{1}$ file remains available as long as in every bucket group there is at most one unavailable bucket. This includes the parity buckets. There is one parity bucket per group. There are also $\lceil M / k \rceil$ groups. Hence:

$$P_1 = ((1-p)^{k+1} + (k+1)p(1-p)^{k})^{\lceil M/k \rceil}.$$

Note that P_1 converges towards 0 when M increases.

An LH*$_{sa}^{2}$ file is available only when at most two buckets in a bucket group are unavailable, including the parity buckets. In every group, there are in practice 2 parity buckets. Hence P_2 yields to:

$$P_2 = ((1-p)^{k+2} + C_1^{k+2}p(1-p)^{k+1} + C_2^{k+2}p^2(1-p)^{k})^{\lceil M/k \rceil}$$

P_2 obviously also converges towards 0. Also, $P_1 > P_2$.

In general, it is easy to see that the availability of LH*$_{sa}^{I}$ file is:

$$P_I = ((1-p)^{k+I} + \sum_{i=1}^{n} C_i^{k+I} p^i (1-p)^{k+I-i})^{\lceil M/k \rceil}$$

Note again that for every I, P_I converges towards 0 when I increases. Also, that P_I increases with I, for any given M.

An LH*$_{sa}$ file gets a new grouping using function $f_{I=:I+1}$ and a new parity file $F_{I>1}$ when it gets bucket $M = k^{I-1}$. From that point every split adds parity records of F_I. The buckets not yet split belong to the groups of $f_{i \le I-1}$ only. The process lasts until M reaches $M = 2k^{I-1}$. Hence, the values $P_1, P_2 .. P_I$ are the reliability P of the LH*$_{sa}$ file for M respectively:

$P = P_1$ for $M \leq k$; $P = P_I$ for $M = k^{I-1}$ .. $2k^{I-1}$.

For values of M between values k^{I-1} and $2k^{I-1}$, every bucket already split by LH function h_{i+1} and the new bucket it creates, participate in f_I groups. The others still participate only in the groups up to f_{I-1}. Hence, for these values of M, one has P monotonously increasing with M from P_{I-1} to P_I. For instance, for $M = k + 3$ buckets $0,1,2,k,k+1,k+2$ will already participate in f_2 groups, while all others will still participate in f_1 groups only. When M reaches $M = 2k$ buckets, the whole file becomes an LH*$_{sa}^2$ file, and remains so until $M = k^2$.

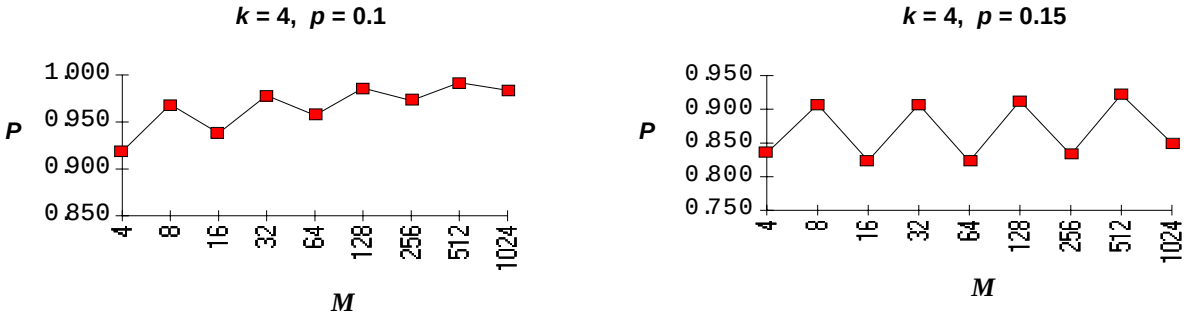


Fig. 4 Uncontrolled reliability of LH*$_{sa}$ files

Fig. 4 shows the scalability of the (uncontrolled) reliability, of two LH*$_{sa}$ files. Both examples assume $k = 4$ but differ with respect to the value of p, set respectively to $p = 0.1$ and $p = 0.15$. The files scale from $M = 4$ to $M = 1024$ buckets. The values of P_I are computed as above. The value of P between P_{I-1} and P_I are linearly interpolated, the difference between both bounds being negligible. The sample values of p have lower availability than in practice, hence making the high-availability difficult to obtain. They mean that a site is unavailable three to four days a month. Such values are orders of magnitude greater than those generally assumed for hardware RAID schemes, [BM92]. They seem nevertheless close to the ones used for a software RAID, e.g., under Windows NT, and other high-availability schemes based on commodity components. An unavailability can occur for various additional reasons, e.g., because of a scheduled maintenance program.

Without any high-availability features, $p = 0.15$ implies $P = 0.5$ already for the case of $M = 4$ bucket file. A 1-availability schema, e.g. LH*$_S$ or LH*$_{g1}$, [L&a97], [LR97], is only slightly better, as $P = 0.7$

already for $M = 8$. The figure shows that, in both cases, LH*$_{sa}$ provides, in contrast, $P > 0.82$ for $p = 0.15$, and $P > 0.92$ for $p = 0.1$ for M up to $M = 1024$. For $p = 0.1$, P is always higher than the availability $(1 - p)$ of a single site. For $p = 0.15$, this is also the case for most of M values. In both cases, P progressively slightly moves up, more for smaller p.

The Appendix shows several curves of P for higher availability values of p, $p = 0.001..0.05$. The values of k are studied between 4 and 128. A larger k is advantageous for access and storage performance as we will soon show. It appears that choosing $k = 4$ leads to the highest and flat P for all p's studied. P decreases for a larger k, but may still provide for an excellent availability, especially for a smaller p. For instance, for $p = 0.001$, even $k = 64$ provides for $P > 0.995$ for the largest file growth studied, i.e., up to $M = 32$K buckets. For $p = 0.001$, even $k = 128$ provides the availability $P > 0.99$ until the file reaches $M = 4$K buckets. However, choosing $k > 4$ for a larger p typically leads to P that is always unacceptably low or decreases rapidly when the file scales up. Thus k should be chosen more carefully for $p > 0.001$.

3.2 Storage occupancy

The storage cost of high-availability, let it be C_s is usually measured as $C_s = S' / S$ where S' is the storage for the data required for the high-availability and S is the storage for the data records. In an LH*$_{sa}$ file F, S' is used for the parity files $F_1 \ldots F_J$ and for additional data, with respect to LH*, within data buckets in F_0 and for the coordinator. These are the file-state data and i value at each F_0 bucket. Their storage cost is negligible. Thus S' is essentially the storage for the parity files that can be measured as the number of parity buckets.

It is easy to infer from the literature, e.g., [H&al94], that for any high-availability schema, static or scalable, with groups of k data records, the minimal s to provide I-availability must be I / k. This value corresponds to I parity records per group. Fewer parity records would trivially compromise I -availability, if a data record had became unavailable with all its parity records.

For an LH*$_{sa}$ file specifically, C_s evolves with the size M of the data file F_0 as follows. When F_0 scales up, s starts from some upper bound U_1, decreases towards a lower bound L_1, then increases again towards some bound U_2, then decreases to some L_2 etc. Each pair U_I and L_I corresponds to I-availability state of F where no F_{I+1}- groups are created yet, i.e., while $I = J$. As proven below, L_I values are $L_I = I / k$ and they

occur for $M = k^I$. With respect to U values, U_1 is $U_1 = 1$ and occurs for $M = 1$. Further U values are defined by more complex expressions. It appears nevertheless that in practice, they can be approximated as:

$U_{I+1} = O(\frac{1}{2} + I/k)$ when $I \geq 1$.

The precision improves when I increases, and for a larger k. The corresponding M values are close to, but above $M = 2\,k^I$. We now prove all these assertions.

Proof. An LH*$_{sa}$ file F is created with one F_0 - bucket and one F_1 - parity bucket. Hence, $C_s = 1$ and $I = J = 1$. Next splits add only F_0 buckets, until F_0 reaches the size of $M = k$ buckets. Hence C_s decreases with each split which trivially means that $M = 1$ leads to $C_s = U_1$, as asserted. The M-th split creates F_0 bucket k, but also 2nd parity bucket of F_1, and 1st parity bucket of F_2, as it starts building *2*-availability, Fig. 2. At this point, one still has $I = 1$, although J just increases to $J = 2$. Also, C_s starts to increase. This proves the asserted existence and value of L_1 at $M = 1 / k$.

To make easier the proof of the general case of $U_{I>1}$, we first derive U_2 and L_2 values. From the file size of M buckets, where $s = L_1$, each of next k splits adds one F_2 – bucket and thus increases C_s. When F_0 reaches $M = 2k$ buckets, all k F_2 – buckets, required for 2-availability, are in place. File F becomes a 2 - availability file, hence I catches up with J, becoming $I = 2$. Split of bucket $2k$ creates one more parity bucket that is 3rd bucket in F_1. Hence C_s still increases. This bucket is required for 2-availability of the "line" of buckets $2k \ldots 3k$-1, Fig. 2. Next $k - 1$ splits do not add any parity bucket. Hence C_s decreases, and there is the local maximum for $M = 2k + 1$:

$U_2 = (k + 3) / (2k + 1) \approx \frac{1}{2} + 1 / k + 1 / 2k = O(\frac{1}{2} + 1 / k)$.

Further expansion till $M = k^2$ adds only one F_1-bucket per k new F_0-buckets. Hence C_s continues to decrease. In contrast k splits following creation of bucket k^2, hence within a new rectangle, create at least 2 parity buckets each, in files F_2 and F_3. Hence, C_s has a local minimum at $M = k^2$ that is:

$L_2 = 2k / k^2 = 2 / k$.

We now prove the L_I values in general; U_I values being addressed next. Observe that the case of $I = 2$ just analyzed generalizes. In fact, for every $M = k^I$, F is an I-availability file were (i) all the necessary parity buckets were built and (ii) last k –1 splits did not create any parity bucket. As it will be addressed more in depth below, C_s value therefore had to decrease from U_I, while M was increasing towards $M = k^I$. As it will appear more in detail below , each of the next splits in a series longer than k^I, starting with the

creation of bucket k^I, begin to build $(I + 1)$-availability so C_s increases again. For F_0 with $M = k^I$ buckets, there are k^{I-1} parity buckets in each file $F_1 ... F_I$. This is the number of corresponding groups in F, Fig. 2. Thus $M = k^I$ realizes locally the best C_s that is:

$C_s = L_I = I\, k^{I-1} / k^I = I / k.$

Finally, we now prove U_I values. Observe from Fig. 2 how the file expands from I-availability file with $M = k^I$ data buckets, towards $(I + 1)$ – availability file ; $I \geq 1$. The goal is to determine the value of M when the last split occurs in the series above, where each split adds parity buckets. This M corresponds to U_{I+1} that C_s reaches while growing from L_I at $M = k^I$. Notice that:

1. Next k^I splits, of buckets $0,1 \ldots k^I - 1$, create k^I parity buckets in F_{I+1}, and k^{I-1} parity buckets per $F_{n \leq I}$, i.e., $k^I + I\, k^{I-1}$ parity buckets in total. These splits lead to $M = 2\, k^I$.
2. Further splits must create $F_{n \leq I}$ buckets for data records that are in F_{I+1} groups, but not in $F_{n \leq I}$ groups that have been created so far. First k^{I-1} such splits, perhaps only one if $I = 1$, create k^{I-1} buckets in F_I. Provided $I \geq 2$, they also create k^{I-2} buckets in F_{I-1}, and k^{I-2} buckets in F_{I-2} for $I \geq 3$ etc. This makes $k^{I-1} + (I - 1)\, k^{I-2}$ new parity buckets in total. The splits also add k^{I-1} data buckets to F_0.
3. If $I = 1$, then Step (2) added a single F_1–bucket, 1st in some line at Fig. 2. The corresponding C_s is $C_s = U_{I+1,}$ since next split does not create any parity bucket. If $I > 1$, thee next k^{I-2} splits create $F_{j \leq I-1}$ buckets for data records that are not in $F_{j \leq I-1}$ groups created so far. This adds $k^{I-2} + (I - 2)\, k^{I-3}$ new parity buckets in total. There are also k^{I-2} more data buckets in F_0.
4. The process loops through Step (3), until a split creates the data bucket needing only an F_1-bucket.

The following formulae for U_{I+1} result from, for any $I \geq 1$:

$U_{I+1} = (I\, k^{I-1} + k^I + I\, k^{I-1} + k^{I-1} + (I - 1)\, k^{I-2} + k^{I-2} + (I - 2)\, k^{I-3} + \ldots + 1 + 1) / (2\, k^I + k^{I-1} + k^{I-2} + \ldots k + 1)$

$= (k^I + (2I + 1)\, k^{I-1} + I\, k^{I-2} + (I - 1)\, k^{I-3} + \ldots + 2) / (2\, k^I + k^{I-1} + k^{I-2} + \ldots k + 1).$

Dropping smaller terms leads to:

$U_{I+1} \approx k^I + (2I + 1)\, k^{I-1} / 2\, k^I$

$= O\,(½ + I / k).$

@

The value of I is a logarithmic function of M. In particular, one can easily see that:

$L_I = \log_k M / k$.

For U_I values, approximating the corresponding M value as $M = 2k^{I-1}$, one gets:

$U_I = O\ (½ + \log_k M / (\log_k 2 + 1)\ k)$.

Thus, the storage cost of LH*$_{sa}$ files scales basically as $O\ (\log_k M)$. Furthermore, U_I relates to L_I as follows:

$U_I = O\ (½ + (I - 1)\ /k) = O\ (L_I + ½ - 1/k)$.

The storage cost C_s thus never exceeds by more than 50% the minimal possible cost, whatever are I and M reached and k chosen (we recall that possible k values are 2,4,8,16....). The minimal cost L_I, reached periodically in $\log_k$ scale when M becomes k^I, achieves in addition the minimal storage for any high-availability schema. Minimizing k to $k = 2$, leads to $U_I = L_I$ and thus to the best storage use for every M with respect to the theoretical minimum. However, such k maximizes globally the number of parity buckets with respect to any larger k. It might not be the wisest choice if the reliability and other costs allow for a larger k. Next larger k ; $k = 4$; leads to the storage cost of at most 25 % above the minimum, also regardless of the file size. Likewise, $k = 8$ leads to the difference of 38 %. Enlarging k further progressively reaches the 50 % variation, but decreases globally the storage for the file. This gain is unfortunately limited since, we recall, a too large k compromises the reliability and the recovery costs. Notwithstanding, the entire behavior of LH*$_{sa}$, as it appears, makes the scheme clearly quite effective with respect to the storage use. The resulting values should usually be acceptable in practice.

To illustrate this point, consider for instance $k = 4$ from Fig. 2. Then, $L_1 = 0.25$ for $M = 4$. Afterwards, the build-up of $I = 2$ starts and C_s re-grows to $U_2 \approx 0.78$ for $M = 9$. Next, C_s decreases back to $L_2 = 0.5$ while I remains $I = 2$, for $M = 16$. Then, $I = 3$ level starts and C_s increases again to $U_3 \approx 1$ for $M = 37$, falling back to $L_3 = 0.75$ for $M = 64$. Afterwards C_s moves up and down between (U_I, L_I) bounds that are (1.25, 1) for I =4 and M up to $M = 256$ data buckets, then (1.5, 1.25) for $I = 5$ up to $M = 1024$ buckets etc. The latter values of each pair, we recall, are the best possible for any I-availability schema.

The choice of $k = 8$ would typically significantly lower C_s for the same M or I, in particular by half for every L_I. However, as Fig. 4 and the Appendix shows such k is acceptable basically only for $p \leq 0.05$. Smaller p value is, larger k can be chosen, e.g., even $k = 128$, as Appendix shows. The storage cost is then

quite negligible. At the expense however of an increased bucket and record recovery costs, as calculated in Sections 3.4 and 3.5 below.

Notice that L_I grows with M, so the importance of the constant ½ in U_I progressively decreases. Thus C_s globally closes on the lowest possible cost while the file scales. The practical incidence of this nice feature seems nevertheless limited to $k = 4$ at best, given the $\log_k M$ growth of I.

3.3 Access performance

In normal mode the parity records do not affect search performance. Hence, key search cost scales as for LH*, i.e., it is two messages typically, and four messages in the worst case, regardless of M. A past unavailability may create an additional forwarding to a new location of a bucket through the coordinator. This adds two messages.

In the degraded mode, when an unavailability is encountered, the search cost includes the record recovery cost, or bucket recovery costs. These much higher costs are addressed below.

The insert cost in the normal mode is the LH* insert cost, typically one message, plus the high-availability penalty that are messages to each F_l ; $l \leq i_m$. A typical insert cost is thus $(1 + J)$. J value scales as $O(\log_k M)$. Thus, the insert cost of an LH*$_{sa}$ file scales well, being a few messages, for even very large files. For instance, for a file with $k = 8$ and reaching 32K buckets, six messages should typically suffice.

Note, that the insert or update penalty must be at least I for any I-availability schema [H&a94]. Depending on the file state, the LH* penalty is therefore either the best or higher by 1 bucket and message. This may seem a small price for the scalability.

The insert cost in degraded mode includes sending the record to the coordinator. The coordinator delivers the record to the correct bucket, perhaps after recovery. This adds two messages.

The additional split cost in normal mode includes typically 2 bJ messages:

b messages to each F_i to remove records from the record groups they participated in.

b messages to each F_i to update parity records of new record groups the records participate in.

When new file F_J starts, the split includes in addition b messages to new file. In the degraded mode, the bucket recovery cost must also be added in. All these costs scale as $O(\log_k M)$, i.e., efficiently.

The cost of a scan in normal mode scales basically as that LH*, for the same reasons as for key search. In the degraded mode, the bucket recovery costs need to be factored in.

3.4 Bucket recovery

Bucket recovery cost, let it be C_b is, at best that of a single bucket, let it be $C_{b,1}$. It's actual value depends on implementation choices for (A6) beyond the scope of this work. A gross evaluation of $C_{b,1}$ may be as follows:

$C_{b,1} = O(k \alpha b / b_m)$.

Here, α denotes the average load factor of a bucket, and b_m denotes the number of records per message from a bucket to the spare site. This site is assumed to carry the entire recovery process. A few messages that must occur between the coordinator and the spare and are not counted explicitly.

The α value depends on the LH* load control policy chosen for F_0 [LNS96]. Typically, one should have $0.7 \leq \alpha \leq 0.85$. In practice, b_m may be guessed within the range [1…10], and b in the order of [100…1000]. Thus, as one could easily guess, C_b is in the order of hundreds of messages at best, and perhaps in the order of thousands.

Assuming $J > 1$, and no race condition addressed in Section 2.4.5, the worst case for C_b is $C_b = O(J C_{b,1})$, so $C_b = O(C_{b,1} \log_k M)$. The worst case cost scales thus efficiently. The derivation of the average C_b is obviously tedious, and not justified here. For our purpose, it suffices to observe that the average should be typically very close to $C_{b,1}$. The probability of 2-failure should be indeed in practice an order of magnitude smaller than that of a single one.

3.5 Record recovery

Record recovery cost, let it be C_r, is clearly typically $O(k)$, and $O(Jk) = O(k \log_k M)$ at worst, assuming no race condition. The exact values depends on the actual implementation of (A8), beyond the scope of this work. The best case can involve even fewer messages, as a record group can contain even a single record, when it is just created. It suffices then to get the parity record only.

The average cost should be usually also about $O(k)$, obviously. Thus the record recovery costs scale efficiently as well.

4 Design variations

One may tune selected performance of the basic schema for some applications. Each optimization comes with some price.

4.1 *High-availability*

The file structure allows, in several cases, for the recovery of more than I unavailabilities. The price is the additional complexity of the recovery algorithms. For instance, these recovery capabilities are easy to see from Fig. 2:

1. From any failure of $(I + 1)$ data buckets for $I > 1$, and, more generally, from any unavailability of $(I + 1)$ buckets other than a data bucket and all its I parity buckets[6].
2. From any failure of k data buckets or records in the same bucket or record group, and from the failure of the parity bucket(s) of that group.

To illustrate the 1st point, consider that one needs to recover buckets 0,1,4,5 while $I = 3$. Bucket 0 can be recovered using F_3 and bucket 16, and the others in group 0 in F_3. Once it is done, one can recover bucket 1 using F_1, then bucket 4 using F_2, and finally bucket 5, using F_1, or F_2. Other orderings are possible.

To illustrate the 2nd point, consider that all four buckets in f_1 - group 0 failed, and that there are no other failures in the file. Then, every data bucket can be recovered using its bucket group in F_2. Once it is done, any parity bucket in F_1 can be recovered as well.

4.2 Bucket recovery

Algorithm (A5) states that all the records from the buckets within the parity group are sent to the spare that performs the recovery of the parity bucket. The actual implementation has to specify the corresponding transfer mode. There are many trade-offs, e.g., between the number of records sent per message and storage efficiency at the spare. These details are left for further work.

It is possible to modify (A5) so that all or most of parity records are recovered at data buckets, instead of being sent to the spare. This would decrease the load on the spare bucket and could make the recovery faster. One solution is that l-th data bucket within the group where $l = r$ mod k, gets all the records with rank r, computes the parity record and sends it to the spare. The price to pay is an increase in the network load, since now the parity records have to be sent through as well.

---

[6] The latter configuration is traditionally called *bad* $(I + 1)$-*erasures* [H&a94], [NW94]. Optimality results proven there for any I-available scheme with $I = 2$ or $I = 3$ that supports all $(I + 1)$-erasures except the bad ones, apply to LH*$_{sa}$ as well.

4.3 Record recovery

Algorithm (A7) does not specify the actual implementation of Q_3 that searches for the parity record with key c, and whose rank r is unknown. To provide for the fast search for the parity record when an insert or an update occurs, it is natural to choose r as the primary key within bucket g_i. The search Q_3 is then basically an intra-bucket scan. If this is too slow for an application, one may add within the parity buckets an additional structure indexing records using the keys c. The price to pay is an additional storage and maintenance of the index during inserts and updates.

4.4 Storage occupancy

The values of C_s higher than L_I during each build-up phase when $k \geq 4$, are the storage cost "premium" one pays to LH*$_{sa}$ schema for its s-availability. We recall from Section 3.2 that these phases start at $M = k^{J-1}$ to incrementally create the I-availability. The rationale behind the premium is that a series of new parity buckets is created serving each less than k data buckets. This behavior opens avenues towards variants of LH*$_{sa}$ tuning the premium, by creating new parity buckets only when the existing ones are used by more data buckets. One may in particular consider to delay the entire I-availability build-up to at least $M = k^I$. This approach has the potential to lower C_s values even for $M = k^l$; l = 2,3... despite the fact that for the basic schema the corresponding L_I values were at their theoretical minimum level. The overall drawback may be a somehow lower reliability. Some of these avenues are explored in next section.

5 Reliability control

The above analysis shows the basic algorithm works fine for a wise choice of k. That is P remains always above some acceptable threshold T, e.g., T being the reliability (1-p) of a single bucket. It also shows that one can tune the scalable availability with this goal in mind. We call such strategies the *reliability control.* Two control variables appear, J and k with the goal variable P to be $P > T$, but also as close as possible to T: The control variable can be manipulated as follows:

New values of J are introduced earlier or later with respect to the uncontrolled breakpoints $N = k^{J-1}$ for new f_J.

Values of k can be decreased or increased when the file expands. That is, when P becomes too close to T, instead of moving to higher J, it may suffice to set k =: k / 2, as long as $k \geq 1$. Afterwards, J can increase,

and k can be increased again to the maximal value such that P would remain the closest possible to T but also above T, to possibly minimize the storage cost.

With respect to the uncontrolled LH*$_{sa}$, one advantage is possibly an improvement of access and storage performance for smaller files. One may achieve also an about a constant reliability for higher p and for larger M. For instance, the analysis shows that for $p = 0.05$, choosing $T = 95$ % requires $k = 4$ or smaller, whether one controls the reliability or not. The control allows J to remain $J = 4$ for up to $M = 8$K. Without it, J reaches $J = 5$ already for $M = 512$. The control saves one access per insert, and $1/k$ of storage. More discussion follows below and in the Appendix.

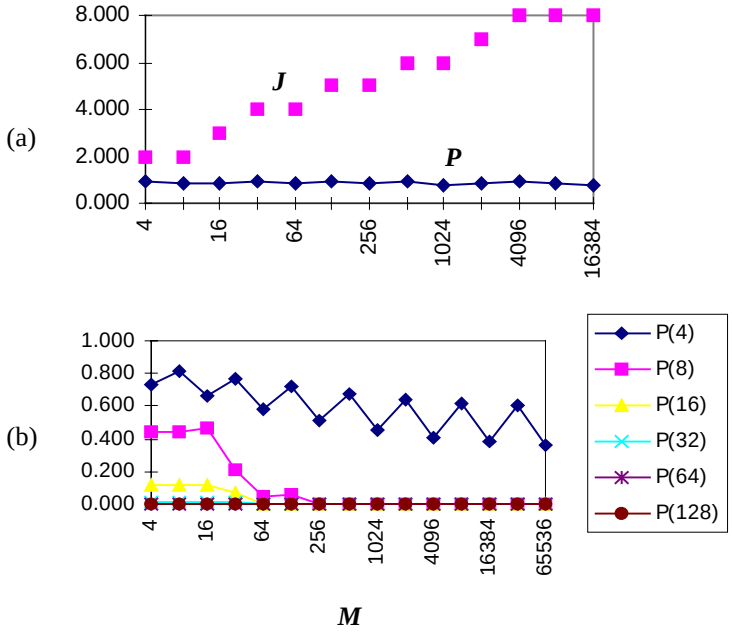
It's also easy to see that one may keep $P > T$ for any M. Any time P decreases too close to T, the control may introduce as may new f_i as needed to raise P enough.

Generally, the decisions to change either J or k offer different trade-offs:

- Increasing J by one leads to one more parity record per data record. Hence it increases by one the insert cost, somewhat the split cost, and by $1 / k$ the storage cost.
- Halving k progressively doubles the number of groups hence of parity records. In contrast it does not add parity records per data record. The advantage is that, unlike above, the insert and split costs do not increase. In contrast, the storage cost becomes substantially higher.

For instance, consider a file that has X records and uses f_3, hence has $3X / k$ parity records, and an insert costs 4 accesses in general. Assume that the availability becomes too close to T while the file is still scaling up, hence either J should increase by one, or $k > 1$ should be halved. First choice means that when the file doubles, there will be $8X / k$ parity records, while the insert cost will increases to 5 accesses in general. If in contrast k is halved, then one ends up with same insert cost, but the file needs the storage for $12X / k$ parity records.

There is therefore no unique algorithm for reliability control. One can control either only J or k or both. Note that the change of k within each group can be incremental. One way for halving it is to start building the new group in addition to the existing one in the bucket group where the split pointer $\tilde{n}$ currently is. Each split adds then records to the new group and removes them from the old one. The new and old parity records are updated in consequence. The old one is removed when the last bucket in it is split. A similar strategy can be used to double k.



(a)

M	*P*	*J*
4	0.901	2
8	0.812	2
16	0.873	3
32	0.920	4
64	0.846	4
128	0.906	5
256	0.822	5
512	0.895	6
1024	0.801	6
2048	0.887	7
4096	0.938	8
8192	0.880	8
16384	0.775	8

Fig. 5 Controlled reliability (a) for k =4, p = 0.2 and T = 0.8, compared to the uncontrolled one (b).

5.1 Fixed group size

One strategy to control the reliability with the group size k fixed is Strategy (S1) that follows. At file creation, one chooses k and T. The file itself is created with M = 1 data bucket. P is controlled every time the split pointer $\tilde{n}$ in the data file points to bucket 0, including the initial state of M = 1. The data file then has $2^{\hat{i}}$ buckets. At this point, the coordinator calculates P using formulae in Section 3. It computes the value P_J that P will have if the file reaches at least k buckets, or, if M doubles again, whichever value is greater. If P_J is under T, then one increases J one by one, until $P_J \geq T$. If J increases, groups are created at each new insert and split according to all new f_i's. . Hence, when split pointer $\tilde{n}$ comes back to $\tilde{n}$ = 0, i. e., when M doubles, all the data records are in new groups. P is calculated again, J is increased if needed, and so on.

(S1) LH*$_{sa}$: Reliability control with fixed *k*

Let M be the current number of buckets in the file, P_J (m) the availability computed for the file of size m according to formulae in Section 3, T the threshold on P, $k = 2^l$ the group size ; l = 1.2... Let $\tilde{n}$ be the split pointer in the data file, and let $\hat{\imath}$ be the file level.

For M = 1 choose the minimal J ; J =1,2.. ; so that P_J (k) > T.

If $\tilde{n}$ = 0, then begin m = max (k, $2^{\hat{\imath}+1}$) ; while P_J (m) < T, J =: J +1 endwhile endif

For every insert, include the new record in all current groups f_J. If J increased, then include at each split all the existing records into new groups.

Step 3 implies the creation or update of the corresponding parity records. Fig. 5 shows an example of the scalability control according to the algorithm above. Fig. 5a shows the evolution of P and of J values for the reliability control. The table shows numerical values corresponding to the graph of controlled P. The probability p that a bucket is unavailable is assumed p = 0.2 which is high. The threshold T is chosen to be T = 0.8 which means that the reliability of the file should be at least that of each bucket. For comparison, Fig. 5b shows the curves P (M) for the uncontrolled reliability. This approach clearly fails for the discussed p. Even k = 4 leads to $P < T$ for already about M = 10 buckets. Controlling the reliability allows in contrast the file to grow as required. The curves and the table show P about constant, up to M = 16K with J reaching J = 8. This would be a very large file. The flat scale-up could clearly continue further using J > 8, to any M needed.

The example shows also that (S1) may provides significant performance gains for smaller files, as J can be significantly smaller. For instance, as long as the file does not exceeds M = 64 buckets, only four parity records are needed, instead of up to eight. This halves the access and the storage costs with respect to those reached for J = 8. That choice would be necessary for any static schema to provide for possible scaling of M between 4K and 16K buckets. As long as this growth does not materialize a useless under-performance would result.

5.2 Variable group size

Varying the group size allows for additional tuning of access and storage performance. The following generalization of (S1) can be one corresponding strategy. It seeks for best access performance, then best

storage performance, provided that $P > T$. The corresponding algorithm is easy enough to explain, without need for a formal definition.

M	*P*	*k*	*i*
4	0.803	2	1
8	0.812	4	2
16	0.802	2	2
32	0.898	2	3
64	0.806	2	3
128	0.815	1	3
256	0.815	2	4
512	0.849	1	4
1024	0.827	2	5
2048	0.877	1	5
4096	0.841	2	6
8192	0.900	1	6
16384	0.811	1	6
32768	0.920	1	7
65536	0.846	1	7

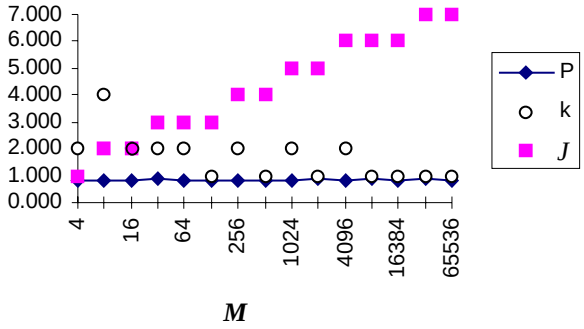


Fig. 6 Controlled reliability with variable *k*, for *p* = 0.2 and *T* = 0.8

The file is created with parameters T and p, and some value $k_{min} \geq 1$. The latter is a user defined minimal bound on k. Using the formulae in Section 3, the coordinator chooses the smallest J, and the largest k so that $P > T$ for $M = k$. When M reaches this value, any time $\tilde{n} = 0$, the reliability is evaluated for the next expansion of the file. However, if the coordinator finds that P would become $P \leq T$ for $M := 2M$, then it first attempts to decrease k before increasing J. This is done by halving k till either $P > T$ or k reaches k_{min}. In the latter case, J is increased as much as needed to again reach P such that $P > T$.

When this occurs, one seeks to increase k through successive doublings of its value, which decreases P as long as P remains $P > T$. This value of k is used for the next expansion. Once $\tilde{n} = 0$ again, the cycle is repeated.

Fig. 6 shows the evolution of P for the same parameters as in Fig. 5: $p = 0.8$, $T = 0.8$, and $k_{min} = 1$. As one could expect, there is some access performance gain with respect to the results for fixed $k = 4$. Especially, for $M \geq 4$K one has $I = 6$ instead of $I = 8$ which represents in general 7 messages per insert instead of 9. These gains are obtained at the expense of the storage for the file. For $M \geq 8$K, since $k = 1$, one now needs the parity records, which are in fact the replicas, up to 6 times the storage of F_0. For fixed $k = 4$, the additional storage needed was only 2 times that of F_0. See the Appendix for a brief discussion of another example.

For some applications, the 3 times difference in the storage, as above, may not justify the gain of two messages per insert, i.e., of 25 %, to access performance. Especially, since these gains correspond only to inserts and updates. In this case one should set up $k_{min} > 1$. One can find also useful the strategies where k_{min} increases during the scale-up, so to increase the relative importance of the storage cost. We leave these strategies for future analysis. Fig. 7 illustrates the simple case of fixed $k_{min} > 1$, set in this example to $k_{min} = 4$, for $p = 0.01$, and T set to $T = 0.95$. Since p is much lower than in the previous example, k changes also much more, between 4 and 32. Such a high variance would make somehow more difficult the user's choice of a fixed k. The overall result is interesting, and, in particular, more efficient than for the corresponding uncontrolled case. To allow for the same availability up to $M = 64$K, one would need to choose $k = 16$, and I would reach $I = 6$, instead of $I = 3$. The uncontrolled reliability would lead therefore to the same storage for the parity files, but to about twice as many messages per insert.

6 Related work

To the best of our knowledge, there were no scalable availability schemes proposed until now. There were in contrast many proposals of 1-availability schemes, and a few providing statically the n-availability with $n > 1$. There were also numerous schemes targeting the scalability without high-availability. These are most of the SDDS schemes referred to in this paper, as well as in [SDDS], aimed at horizontal scalability, at network and switched multicomputers, [KLR96], as well and some others, e.g., [SPW90], and [JK93] among the earliest, targeting multiprocessor supercomputers. Other proposals seek vertical scalability, over

a fixed and rather small size multicomputer, e,g,, 16 nodes, [B&al95], [FBW97], [M97a] and [P97]. Some of the implementations provide support for high-availability through mirroring, [B&al], [H96]. Others propose a more or less delayed replication for disaster recovery, and lower level capabilities, e.g., for fail-over recovery [M96], [M97].

M	*P*	*k*	*J*
4	0.957	32	1
8	0.957	32	1
16	0.957	32	1
32	0.957	32	1
64	0.952	16	1
128	0.946	8	1
256	0.963	32	2
512	0.977	16	2
1024	0.954	16	2
2048	0.971	8	2
4096	0.980	4	2
8192	0.961	4	2
16384	0.965	16	3
32768	0.987	8	3
65536	0.975	8	3

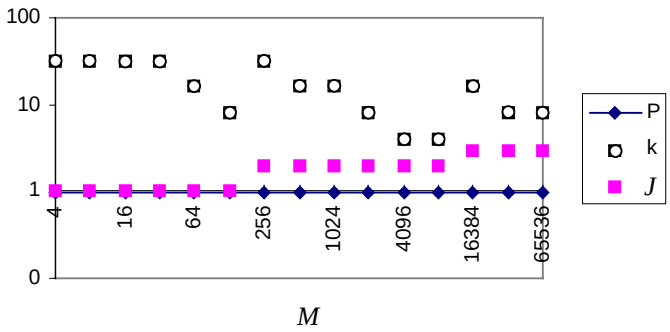


Fig. 7 Controlled reliability with variable *k*, for *p* = 0.01 and *T* = 0.95

In fact, many schemes using mirroring were investigated in the seventies and eighties. Their goal was however high search performance rather than high-availability. Their prime disadvantage is the high storage cost. This limits the approach in practice to files providing 1-availability at best. The application of mirroring to scalable multicomputer files was studied in [ChS92], and, through the LH*$_m$ schema in [LN96a], for LH* files specifically.

The research in the nineties on 1-availability schemes was centered on variants of RAID schemes [PGK88]. These schemes were much more storage efficient than mirroring, at the expense of access performance. Neither the scalability, nor the multicomputer environment was the target for that work. There were nevertheless attempts towards the latter goal, e.g., [SS90], [MLC93]. An on-going research project, the DIY-RAID system, addresses the scalability of RAID schemes more specifically [A&al97]. The goal is a thousand-node RAID configuration. Nevertheless, the system aims to provide 1-availability, or 2-availability at best, using the "orthogonal RAID" approach [A&al97].

The need for *n*-availability schemes for RAID systems with $n > 1$ was observed early. In [G&al89] it was suggested that basic RAID schema could be made *n*-dimensional (orthogonal) for this purpose. As Fig. 2 shows, LH*$_{sa}$ is somehow rooted in this approach, every k^2 sites forming a rectangle with horizontal and vertical parity. In [BM92] one proposes an efficient *n*-availability scheme using the MDS codes. The EvenOdd schema in [BBM93] provides particularly efficiently for 2-availability. Some linear coding techniques, for 2-availability, or possibly for 3-availability, are also addressed in [H&a94]. In [NW94], there are complementary proposals for cases of correlated disk failures that may generalize to LH*$_{sa}$ bucket failures. In [T98], there is an overview of several RAID oriented schemes seeking for 2-availability. Recently, the DATUM RAID scheme provides for *n*-availability in an optimal amount of redundant storage [ABC97]. All these schemes address the static *n*.

The already mentioned LH*$_m$, as well as LH*$_s$ and LH*$_g$ schemes, [L&a97], [LR97], were first to address the high-availability requirement in the context of horizontal scalability. One basic difference to the traditional and distributed RAID schemes is a more elaborated logical and software addressing schema, for both data and parity records. RAID schemes basically use physical mappings, like the round-robin addressing. Such schemes, behind the controller, or in software on a multi-disk PC, e.g., with Windows NT, are simple but cumbersome for scalability. In the LH* schemes, the data and parity records have keys, and dynamic addresses calculated at the client and servers using the LH* access method. This feature is crucial for scalability. As it appeared in particular for LH*$_{sa}$, both data and parity records may incrementally move to an increasing number of locations.

LH*$_s$ applies the striping and provides for 1-availability. Its disadvantage is that record scans become more expensive than in LH*. LH*$_g$ uses record grouping, as does LH*$_{sa}$, making the scans as efficient as

for LH*. It provides basically for 1-availability with the split cost lower than that of LH*$_{sa}$. It is also shown in [LR97] how this schema can be expanded to provide for 2-availability.

The discussion of LH*$_m$ in [LN96a] points out to some basic types of high-availability LH* schemes. A scheme with the *structurally similar* data and parity files groups in one bucket the parity records of data records in the same bucket group. A schema with *structurally dissimilar* data and parity files, hashes in contrast these parity records over the parity file. Both types of schemes have advantages and inconveniences. The structurally similar schemes tend to be more efficient during normal and degraded use. The structurally dissimilar schemes typically loose less data if a catastrophic failure occurs, e.g. of more than *I* buckets simultaneously. The difference can be the loss of a few records only, instead of a whole bucket. The LH*$_{sa}$ schemes above are of the structurally similar type. LH*$_g$ schema is of the structurally dissimilar type. Design of a structurally dissimilar LH*$_{sa}$ schema remains a future work.

7 Conclusion

Scalable distributed or parallel storage systems have become a major trend in research and industry. These schemes require high-availability and it is desirable to make this feature scalable as well. LH*$_{sa}$ schema provides this property that cannot be achieved by any static *n*-availability scheme. It opens new perspectives for high-availability applications and the use of large scalable files in general. Future work should explore deeper scalability analysis of the uncontrolled and controlled LH*$_{sa}$ variants, through formal calculus, and simulation. One should also analyze more in depth LH*$_{sa}$ variants. One should furthermore experiment with the implementations. Finally, one should design other *s*-availability schemes.

More generally, one may observe that the traditional data structures were designed for efficient data addressing and storage. High availability was a feature of lower levels of a storage system. This dichotomy appears inefficient for building the storage systems for the scalable and distributed world. High-availability should be a feature of the data structures, as important as the traditional ones.

References

[A&al95] Agerwala & al. SP2 System Architecture. IBM Syst. Journal, 34, 2, 1995. 152-184.

[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.

[A&al97] Asami, S. & al. The Design of Large-Scale, Do-It-Yourself RAIDs. CS Tech. Rep. UC Berkeley, 1997.

[B&al] Boloski & al. The Tiger Video Server. www.research.microsoft.com

[BBM93] Blaum, M., Bruck, J., Menon, J. Evenodd: an Optimal Scheme for Tolerating Double Disk Failures in RAID Architectures. IBM Comp. Sc. Res. Rep., (Sep. 1993), 11.

[BM92] Burkhard, W., Menon, J. MDS Disk Array Reliability. UCSD Res. Rep. CS92-269, 26.

[B&al95] Baru, W., C., & al. DB2 Parallel Edition. IBM Syst. Journal, 34, 2, 1995. 292-322.

[C&al95] Cabrera, L. et al. ADSM: A Multi-Platform, Scalable, Backup and Archive Mass Storage System. IEEE-COMPCON-95, IEEE Press, 1995, 420-427.

[ChS92] Chamberlin, D., Schmuck, F. Dynamic Data Distribution (D^3) in a Shared-Nothing Multiprocessor Data Store. *VLDB-92*, 1992.

[FBW97] Freedman, C., Burger, J., DeWitt, D. SPIFFI -- A Scalable Parallel File System for the Intel Paragon. Trans. on Par. and Distr. Syst.

[G&al89] Gibson, G & al. Coding techniques for handling failures in large disk arrays. Intl. Conf. On Arch. Support for Prog. Lang. And Op. Syst., 1989, 123-132.

[H&a94] Hellerstein, L, Gibson, G., Karp, R., Katz, R. Patterson, D. Coding Techniques for Handling Failures in Large Disk Arrays. Algorithmica, 1994, 12, 182-208.

[H96] Haskin, R. Schmuck, F. The Tiger Shark File System. COMPCON-96, 1996.

[I98] Inktomi Corporation. http://www.inktomi.com/

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

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

[KH98] Kumar, V., Hsu, M. (Ed.). RECOVERY MECHANISMS in DATABASE SYSTEMS. Prentice Hall, 1998, 989.

[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.

[L80] Litwin, W. Linear Hashing : a new tool for file and tables addressing. Reprint from Intl. Conf. On Very Large Databases, VLDB-80 in READINGS IN DATABASES. 2-nd ed. M. Stonebraker , M.(Ed.). Morgan Kaufmann Publishers, Inc., 1994.

[L96] Lomet, D. Replicated Indexes for Distributed Data IEEE Intl. Conf. on Par. & Distr. Systems, PDIS-96, (Dec. 1996).

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

[L&a97] Litwin, W., Neimat, M-A., Levy, G., Ndiaye, S., Seck. T. LH*$_S$: a high-availability and high-security Scalable Distributed Data Structure. IEEE Workshop on 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). Submitted.

[MLC93] Montague, B., Long, D., Cabrera, L. SWIFT/RAID A Distributed Raid System. IBM Res. Rep. RJ 9501, 1993, 25.

[M94] Menon, J. Performance of RAID5 Disk Arrays with Read and Write Caching. Distr. & Par. Databases, 2, 1994, 261-293.

[MRW95] Menon, J., Riegel, J., Wyllie, J. Algorithms for Software and Low-cost Hardware RAIDs. IEEE-COMPCON-95, IEEE Press, 1995, 411-418.

[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.

[M97] Microsoft SQL Server Scalability. A White Paper from the Desktop and Business Systems Division. Microsoft, 1997, 27.

[M97a] Clustering Support for Microsoft SQL Server. White Paper, May 1997, 16.

[M97b] Two Commodity Scaleable Servers: A Billion Transactions per Day and the Terra-Server. White Paper, Desktop and Business Syst. Div. May 1997, 27.

[M97c] SQL Server VLM. Microsoft Scalability Day. http://204.203.124.10/backoffice/scalability/coverage.htm [P97] Patel, J & al. Building A Scalable GeoSpatial Database System: Technology, Implementation, and EvaluationACM-Sigmod, 1997, 336-347.

[M98] Menon, J. A Performance Comparison of RAID5 and Log-Structured Arrays. In RECOVERY MECHANISMS in DATABASE SYSTEMS. Kumar, V., Hsu, M. (Ed.). Prentice Hall, 1998, 989.

[NW94] Newberg, L., Wolfe, D. String Layouts for Redundant Array of Inexpensive Disks. Algorithmica, 1994, 12, 209-224.

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

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

[RM96] Riegel, J. Menon, J. Performance of Recovery Time Improvement Algorithms for Software RAIDs. IEEE Conf. On Parallel and Distr. Database Systems (PDIS-97). IEEE-Press, 1997, 56-65.

[R97] RAMAC™ Scalable Array Storage 2. www.almaden.ibm.com/storage/hardsoft/diskdrls/scalable/sca2spec.htm

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

[SDDS] SDDS-bibliography. http://192.134.119.81/SDDS-bibliograhie.html

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

[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.

[T98] Thomasian, A. RAID5 Disk Arrays and Their Performance Evaluation. RECOVERY MECHANISMS in DATABASE SYSTEMS. Kumar, V., Hsu, M. (Ed.). Prentice Hall, 1998, 989.

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

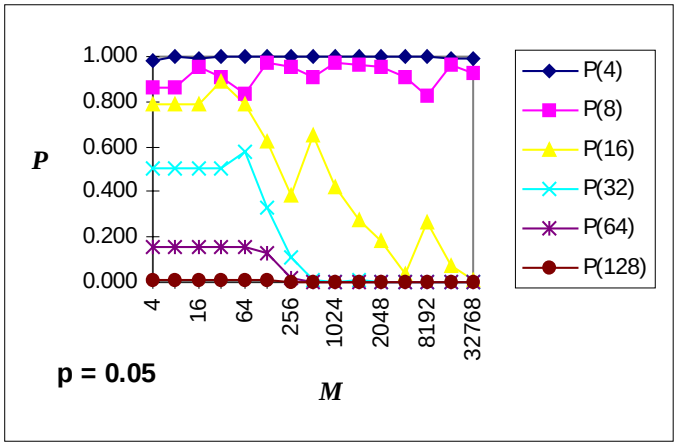
[U94] Ullman, J. New Frontiers in Database System Research. *Future Tendencies in Computer Science, Control, and Applied Mathematics.* Lecture Notes in Computer Science 653, Springer-Verlag, 1994. A. Bensoussan, J. P. Verjus, ed. 87-101.

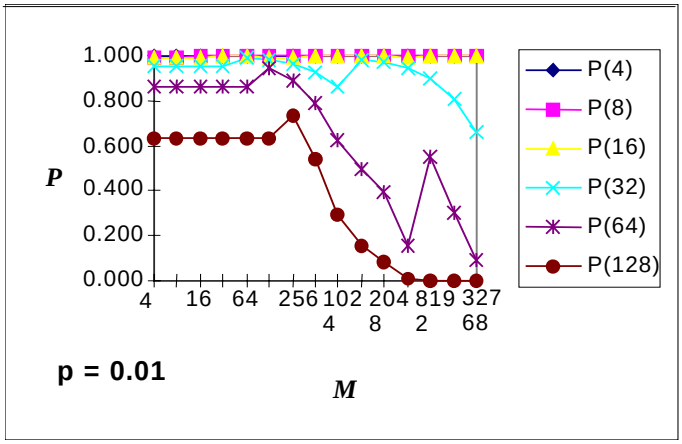
[W96] Wilkes, J. & al.. The HP AutoRAID hierarchical storage system. ACM-TCS, 14, 1, 1996.

Appendix

The curves below complete Fig. 4 and show the uncontrolled reliability $P(k)$ of LH_{sa} for various practical values of p and of k and for a file scaling up to M = 32K buckets (sites). The scale of M-axis is logarithmic. Larger p requires smaller k to keep the availability scalable. For $p = 0.05$ $k = 8$ suffices if the availability $P > T = 0.85$ suffices. For a higher T, $k = 4$ should be chosen. For $k > 4$, P tends towards zero and the availability is not scalable anymore. For smaller p, especially for p =0,001, more and more values of k under analysis, which are k = 4,8,16,32,64,128 provide the scalability, at least to some practical M < 32K. For instance, for $p = 0.005$, if $T = 0.85$ suffices, then $k = 128$ suffices up to $M = 512$ buckets. This is already a large value by today's needs.

The curves show also the utility of the reliability control. For instance, assume that $T = 0.85$ for $p = 0.01$. Then it suffices to start the file with $k = 64$. If M exceeds $M = 256$, k should be halved. If M exceeds further 8K, it should be halved again. The uncontrolled reliability requires $k = 16$. It leads thus to up to four times more storage for smaller M' and to a higher insert and split costs, as i progresses faster as well.





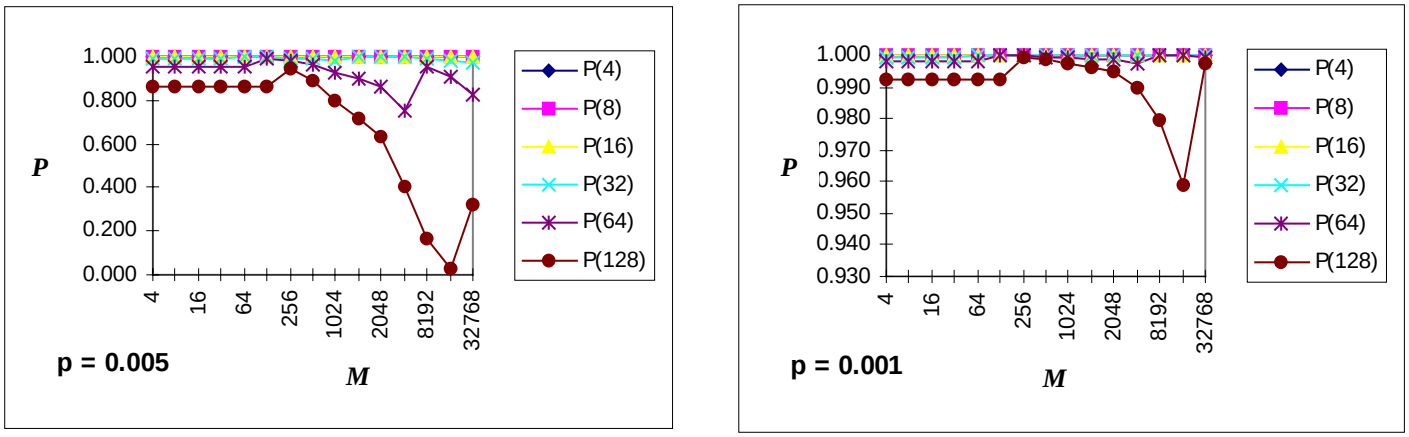


Fig. 8 Uncontrolled reliability of LH*$_{sa}$ files for various values of *k* and of *p*.

Glossary of terms and symbols

a – LH address for a key in an LH* file. Also called correct address.

a' – client image of a (may be different from a)

b – bucket capacity

B – parity bit

c – record key (identifier), hashed by LH-functions

C_s – high-availability storage cost

C_b – bucket recovery cost

C_r – record recovery cost

f – grouping function

F – LH*$_{sa}$ file comprising the data file F_0 and the parity files $F_1 \ldots F_J$

F_0 – LH*$_{sa}$ file with (user) data records

F_i – LH*$_{sa}$ i-th parity file

g – bucket (availability) group number

(g, r) – record with rank r (availability) group number

h – LH-function

i – bucket availability level, also called grouping level

$\hat{i}$ – LH* or LH*$_{sa}$ file level

i' – client image of LH* or LH*$_{sa}$ file level

I – LH*$_{sa}$ (current) availability level, making the file I-available

IAM – image adjustment message

J – (current) number of parity files in an LH*$_{sa}$ file, making it J-available, or $(J – 1)$-available

k – maximal size of LH*$_{sa}$ bucket and record availability groups

LH – Linear Hashing

LH* – (Linear Hashing)*

LH*$_{sa}$ – LH* with s-availability

M – LH*$_{sa}$ number of data buckets (in F_0 file)

n-availability – file capability to remain available despite unavailability of any *n* buckets

N – initial number of buckets in an LH or LH* file.

$\tilde{n}$ – split pointer

n' – client image of split pointer

p – probability that a bucket is unavailable (failed)

P – probability that (entire) file is available (file reliability)

r –record rank in a bucket

RAID –Redundant Arrays of Independent Disks (originally: of Inexpensive Disks)

s-availability – scalable availability

T – threshold for an LH*$_{sa}$ file with controlled reliability