Design Issues For Scalable Availability LH* Schemes with Record Grouping[1]

Witold Litwin

U. Paris 9, France

Jai Menon

IBM Almaden Res. Cntr., San Jose CA, USA

Tore Risch

U. of Uppsala, Sweden

Thomas J.E. Schwarz, S.J.

Jesuit School of Theology, Berkeley, CA, USA

Abstract

LH* schema is among most studied Scalable Distributed Data Structures. LH* variants have been in particular developed for the high-availability files, capable of serving all the data despite unavailability of some storage sites. The scalable availability schemes, tolerating increasingly more failures when the file grows, are of particular importance. We present three high-availability LH* schemes using new concept of record grouping. We discuss the common building blocks and the specific features of each schema. We compare the design issues, properties and performance.

1 Introduction

Multicomputer scalability, distributed and parallel access, and high-availability became buzzwords [M96], [M97c], [M97b]. However, to satisfy all these goals jointly is not a simple task. The concept of a Scalable Distributed Data Structure (SDDS) was introduced in [LNS93] to respond to first two goals. Many SDDSs have been proposed [SDDS]. Some proposals concern the *high-availability SDDSs*, able to serve any data stored despite unavailability (failure, inaccessibility…) of some storage sites (nodes). Research showed that direct adaptation of known recipes for high-availability to an SDDS may not be practical. Mirroring may cost too much [ChS92], [LN96a]. The protection against unavailability of even a single storage site of an SDDS requires doubling the number of storage sites; a substantial cost for a larger multicomputer, e.g. the 166–site multicomputer of Inktomi at Santa Clara, CA, or the 100-site one of Yahoo in Vienna, VA. More generally, to protect against the simultaneous failure of *n* storage sites requires *n* times more sites. Distributed striping of data records as in RAID schemes, e.g., like in [SS90], [MLC93] or [L&a97], greatly reduces the storage overhead. But it also scrambles the logical records, prohibiting parallel scans and the function (query) shipping that filter the data directly at each storage computer. These capabilities are on the other hand among first needs for a modern database. The completion of a query with aggregate functions could otherwise easily take days, especially on a larger database like the 13 TB database of UPS.

Not withstanding, databases increasingly require the high-availability. Especially the widely used ones like those of Web servers, e.g., E-Bay, or of financial institutions. They usually follow the 24/7 regime. The cost of 22 hour-long crash of E-Bay on July 9, 1999, was valued at almost 4 B$ in lost market value, and 25 M$ of operating loss [B99a]. A minute of unavailability of a financial database costs between 10 and 27 K$. No wonder that these figures propel the high-availability among most important design goals.

We present below one approach to an SDDS design that reconciles the discussed goals. We called it *record grouping* [LR97]. A record group involves $k >> 1$ data records, each from a different data storage site (*data bucket*). Each group has $n \geq 1$ parity records, each in a different *parity bucket*. One may recover up to any *n* unavailable records in the group. The *n*-availability starts with $n = 1$ and scales incrementally with the file. Such *scalable* (high)-availability is necessary to keep the reliability (the

[1] DIMACS Workshop on Distributed Data and Structures, Princeton U., (May 1999), **Carleton Scientific**, (publ.) 1999.

probability that all the data records are available for the application) from decreasing when the file scales.

Making an SDDS highly available through the record grouping, instead the mirroring, leads to a much smaller storage overhead. The order is *n*/*k*, instead of *n*, e.g., 12.5 % instead of 200% for a 2-available file. In fact, *k* can be arbitrarily large, making the storage overhead arbitrarily small, although this increases in turn the record recovery cost linearly. With respect to the record striping that scatters the records into fragments, the grouping leaves the records entire and where they were. As long as no failure occurs, the key search and the (parallel) file scan performance of the SDDS remain unaffected. Only the access performance is altered, and only for data modifications, by messages sent to buckets with parity records.

Three SDDS schemes using the grouping, termed $LH*_g$, $LH*_{SA}$, and $LH*_{RS}$ have been investigated until now. All use the same kernel SDDS termed LH* [LNS93]. They differ by the group structure. $LH*_g$ uses location independent grouping. A record remains in the same group despite its moves to new sites when splits occur. $LH*_{SA}$ and $LH*_{RS}$ use location depended grouping. Any split changes the group of any record it moves. In $LH*_{SA}$, *n*-availability of the file results from the structure where a record belongs to *n* or (*n*+1) different groups. Each group recovers from a single site failure, using one parity bucket. In $LH*_{RS}$ and in $LH*_g$, a record belongs to only one group, which has *n* or (*n*+1) parity buckets. The computation of parity data uses the Reed Salomon codes (RS-codes). Each schema has somehow different properties. The diversity should fit different applications.

The schemes are described in depth in dedicated papers, [LR97], [LMR98], [LS99]. One exception is the $LH*_g$ that is here somehow redesigned for *n*-availability with RS-codes. Below, we overview these schemes, centering on similarities and differences. Section 2 recalls the principles of SDDSs. Section 3 describes more in depth the record grouping and our use of RS-codes. Section 4 discusses the parity data management. The comparative analysis is in Section 5. Section 6 contains the conclusion.

2 Scalable Distributed Data Structures

2.1 General principles

Scalable Distributed Data Structures (SDDSs) form a new class of data structures specifically for multicomputers. These can be *switched* multicomputers, using some bus to interconnect multiple CPUs, e.g., IBM SP2 [A&al95]. More often, these are *network* multicomputers, interconnecting mass produced PCs, WSs, and servers through popular high-speed networks, e.g., Ethernet, [T95]. An SDDS file consists of data records identified each by at least one key, Figure 1. Records are stored in buckets on server sites, called *servers*. The number of servers scales with insertions through the splits of overflowing buckets. Each bucket is identified by number 0,1,… or by the cube enclosing keys it may store. The file is partitioned onto the servers, e.g., through hash or range partitioning.

Records are stored according to some global addressing rule. The rule can be known to a particular component called the *coordinator* or is enforced in distributed way. If there is a coordinator, it manages also the splits. Overflowing buckets alert the coordinator using messages. The coordinator chooses the bucket to split, according to the addressing rule. The rule parameters change dynamically according to the current file size.

SDDS *clients* that reside on the application sites access the servers. The clients are autonomous, not always available for access from the servers, and can be mobile. The file manipulation requests are typically a key search, a parallel scan, an insert of a record, an update or a deletion. Each client has its own image of the addressing rule that it uses to send requests. This avoids a centralized directory that could create a hot spot. The parameters of the image may be outdated, as clients are not synchronously aware of the splits. The client may send the request to an incorrect server. A characteristic property of any SDDS is that any server autonomously determines whether it constitutes the correct one for the request. If not it forwards the request to a presumably correct one. That server may still be an incorrect one, forwarding the record again etc. Known SDDSs localize the correct server in a few hops at worst.

Iff the forwarding happens, the client receives an *Image Adjustment* Message (IAM). This message contains data actualizing the client image. It is required for any SDDS that these data are such that no addressing error occurs twice, as long as the file does not scale.

2.2 The LH* SDDS

These general principles were applied to design various SDDSs [SDDS]. As traditional data structures, they provide now for hash files, primary key ordered files, and for multi-key files. The LH* hashing schema was the 1st proposed and is best known, [LN96a], [K98], [B99]. It uses the well-known linear hashing, [L80], as the global rule, and for the client images. Accordingly, the splits of the buckets numbered 0,1… are performed in the deterministic order 0,0,1,0,1,2,3,0…2^i – 1,0…. LH* hash functions work so that each split of bucket *m* adds bucket $m + 2^i$ to the file with last current bucket being bucket $m + 2^i –1$ [B99]. The *file level i*, and the pointer to next bucket to split, usually noted *n*, constitute the *file state* data of LH*. They are used for hashing and determine the correct location of any key *c*. The coordinator maintains them.

An LH* client does not access the file state data, but uses to locate *c* through LH* hashing algorithm, its own image (*i' n'*), not necessarily the actual one. For a new client, $i' = n' = 0$. In the case of an addressing error, the content of the IAM allows to refresh (*i' n'*) enough to find henceforward *c* correctly, provided it did not move because of a split in the meantime. Updated values are not necessarily the correct file state.

Nevertheless, it appeared that most key searches in an LH* file, and most inserts into, are completed without any forwarding. One also proved that two hops suffice at worst, regardless of the client image, and of the file size. Several variants of LH* were proposed, some without the coordinator, and some with the high-availability. Two structures do not use the grouping. The $LH*_M$ uses mirroring and provides 1-availability. The $LH*_S$ uses record striping and parity stripes, also for 1-availability. Its storage overhead is much smaller than that of $LH*_M$, but no parallel scans are possible. On the other side, its scattering of stripes enhances data security.

The record grouping was applied to three LH* schemes already mentioned in the introduction. All these schemes have a coordinator, expanded with the functions for the availability management. Their design principles for the high-availability are as follows.

3 High-availability through record grouping

3.1 Record groups

Let *F* be an SDDS file without high-availability features with records created by the application. Below, *F* is called the *data* file and its records are called *data* records. Each data record is uniquely characterized within its bucket by its *rank*, basically its position within the bucket. A *record group* is a set of $m \geq 1$ data records with the same rank each from a different bucket. Each record group is provided with $n \geq 1$ *parity* records. Multiple parity records for a group are specifically ordered and referred to as 1st, 2nd…. Each parity record is stored in a different parity bucket. Parity buckets form one or more *parity* files. A *high-availability* SDDS file with record groups consists of *F* with records groups and of its parity file(s).

Groups are formed using some *grouping* functions. The basic grouping function used in all the discussed schemes is $g = m(c) \bmod k$. Here, *g* denotes the group number, *c* denotes the key of the data record, *k* is some file parameter called the group *size*, and *m*(*c*) is the address:

- Where the data record *c* is currently stored, for $LH*_{SA}$ and $LH*_{RS}$,
- Or, where the record *c* was inserted, for $LH*_g$.

The $LH*_g$ and $LH*_{RS}$ schemes use only this function. $LH*_{SA}$ uses a family of grouping functions that we will address below. It follows from the above for $LH*_{RS}$ and $LH*_{SA}$, that when a split moves a data record, its group typically changes, and the parity records have to be updated. In contrast, for $LH*_g$, the record groups are location independent, and so splits are cheaper. This is possible only because of the property of the LH* schema that no split ever moves into the same bucket records inserted into different ones.

Figure 1 shows the structure of data and of parity records. A data record has a key usually noted *c* and some non-key data. A parity record is identified by its own key, usually noted *p*. Key *p* is for $LH*_{RS}$ and $LH*_{SA}$ the rank *r* of the data records forming the group. For $LH*_g$, $p = (g, r)$.

Next field of a parity record consists of keys $c_1…c_m$ of the group members, $m \leq k$. These are used to search for the available members when an unavailable one should be recovered. Finally, there are encoded parity data. These encode only the non-key data within the data records. The length, in bits, or bytes, of the parity data field is that of the largest non-key field in the group.

For LH*$_g$ and LH*$_{RS}$, a group can have $n > 1$ parity records, making the group n-available. In other words, one may recover any group member despite the unavailability of up to any n data or parity records of the group. All the parity records of a group share the key p. We recall that they are nevertheless in different buckets. Otherwise, the n-availability would not be possible.

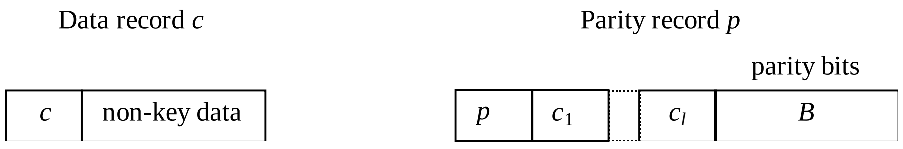

Figure 1 Record structures in LH* files with record grouping

For LH*$_{SA}$, a group has a single parity bucket that makes the group 1-available. The n-availability of the data file results from the membership of each record in at least n different groups. The groups sharing a record are formed so they intersect at only that member. For instance, assuming $k = 4$, a data record in bucket $m = 0$ may be involved in the following groups with the data records with the same ranks in other buckets m. The groups are created successively when the file scales:

(0,1,2,3), (0,4,8,12), (0,16,32,48), (0,64,128,192)…

A record in bucket 1 participates then in the groups:

(0,1,2,3), (1,5,9,13), (1,17,33,49), (0,65,129,193)…

The general formulae for LH*$_{SA}$ grouping functions are in [LMR98].

In this schema, if more than one member of a group is unavailable, all but one are recovered using other groups they belong to. For instance, if buckets 0 and 1 are unavailable in our example, then a record in bucket 1 is recoverable from group (1,5,9,13), which makes the record in its group in bucket 0 recoverable within its group (0,1,2,3). Provided there is no more than n records unavailable in the group, or, more generally, at most n unavailable data or parity buckets in the file, there are always enough groups where each of them is the only one unavailable [LMR98].

Different grouping schemes lead to different access, recovery and storage performance, broadening the class of applications that may benefit from. They share however the common major advantage that the addressing scheme of the data file remains unaltered by the high-availability adds-on. Every data record is stored at the same address it would be without the parity data. The key and parallel scan search performance remain unaffected in the *normal* mode, i.e., when no unavailability occurs. Only file modifications require additional messaging towards the parity buckets.

3.2 Parity Calculus

3.2.1 Operations on Galois Fields

For parity calculus, data and the parity formally consist of symbols, bits or l-bit bytes, $l = 4$ or $l = 8$ in practice. The three schemes use the addition and, for LH*$_g$ and LH*$_{RS}$, the multiplication in the Galois Field whose elements are all the values of symbols used, i.e., $GF(2)$ or $GF(2^l)$. More generally, these two schemes use these operations to compute the codewords of a linear Reed Solomon code (RS-code). The size of the field has some importance. The total number of data and parity records in a group cannot exceed $2^l + 1$. The 4-bit symbols suffice for many applications; the 8-bit ones suffice probably almost always at present.

We recall that a Galois Field $GF(N)$ is a set with N elements (symbols) and the operations of addition, and multiplication, as well as their inverses that are subtraction and division. There are two distinguished elements, a zero element, noted 0, and a one element, noted 1, in the set. The operations over $GF(N)$ pose the usual properties of their analogues in the real numbers including those properties referring to 0 and 1. Furthermore, regardless of the size of the field, both the addition and the subtraction, boil down to the well-known bitwise XORing of the symbols. For $GF(2^4)$ for instance:

$x = 1010$ and $y = 1011 \Rightarrow z = x+y = 1010 \text{ XOR } 1011 = 0001$.

This happens incidentally to be 1 of this *GF*, as shown at Table 1 below.

In LH*$_{SA}$, this addition (subtraction) is the only operation used for encoding of the parity symbols, and for decoding to recover the unavailable member of the group. More precisely, each parity symbol *P* is:

$$P = D_1 \text{ XOR} \ldots \text{XOR } D_m.$$

Here *D* is the data symbol with the same offset within each group member, assumed broke to successive symbols in its non-key part. Likewise, the unavailable symbol D_l; let it be $1 < l < m$; is computed for recovery as:

$$P = D_1 \text{ XOR } D_2 \ldots \text{ XOR } D_{l-1} \text{ XOR } D_{l+1} \ldots \text{XOR } P,$$

The multiplication in a Galois Field, applied in the two other schemes, is generally more involved. We now concentrate on these schemes, until the contrary is stated. A practical way, assumed for the discussed schemes, is to use the *GF* log and antilog tables. These tables are available in the open literature. For every element *x* of a *GF,* the tables contain $\log_\alpha x$ which is the corresponding power of a *primitive* element α. We recall that α is the element such that every other non-zero element of the *GF* is some different power of α and that every *GF* has at least one α. The following holds then in any $GF(2^l)$ for any elements *x*,*y*,*z* where + denotes the “usual” addition modulo 2^l - 1:

$$z = xy = antilog_\alpha (\log_\alpha (x) + log_\alpha (y))$$

To multiply two symbols, one therefore needs to locate their logs, add them modulo 2^l - 1, and finally locate the symbol facing this antilog. Each table contains only 2^l elements, i.e., at most 256 in our case, and can be searched very efficiently.

As an example, Table 1 shows the log table for $GF(2^4)$, i.e., for 4-bit long symbols. Each symbol has three practical representations. The log table for $GF(2^8)$ are, e.g., in [LC93].

string	int	hex	log
0000	0	0	**-∞**
0001	1	1	**0**
0010	2	2	**1**
0011	3	3	**4**
0100	4	4	**2**
0101	5	5	**8**
0110	6	6	**5**
0111	7	7	**10**
1000	8	8	**3**
1001	9	9	**14**
1010	10	A	**9**
1011	11	B	**7**
1100	12	C	**6**
1101	13	D	**13**
1110	14	E	**11**
1111	15	F	**12**

Table 1: Log table for $GF(2^4)$.

3.2.2 Parity Encoding Schema using RS-codes

We now call *segment* the maximal possible number of records in a group together with the maximal number of the parity records for the symbol size. We assume formally that all these records exist, some being simply dummy, i.e., full of 0s. We use *s* for the segment size, *k* for the record group size, and *n* for the number of parity records. Thus, $s = k + n$, and the group is *k*-available. Records consist of already discussed *l*-bit symbols, $l = 4,8$.

Each parity record is generated one symbol at a time from the *m* data symbols $a_1,a_2,a_3...a_k$ with the same offset, with a_i within the ith record in the group. The data symbols form vector **a** = $(a_1,a_2,a_3...a_k)$. All the data and parity symbols of the segment form vector **u** = $(a_1,a_2...a_k,a_{k+1}...a_s)$, which is the codeword. The first *k* coordinates of **u** are the coordinates of **a.** The remaining *n* coordinates of **u** are all the corresponding parity symbols.

We obtain **u** by multiplying **a** by the linear (systematic) RS-code generator matrix **G**; namely **u** = **aG**. **G** has *k* rows and *s* columns. Its coordinates are elements in $GF(2^l)$. It results from the theory of linear RS-codes that **G** consists of two concatenated sub matrixes; namely **G** = **I**|**P**. Matrix **I** is a *k* x *k* identity matrix, hence **aI = a**. That is why first *k* coordinates of **u** form **a**. Only the columns of matrix **P** operationally contribute to the *n* parity symbols. Each parity symbol within *i*th parity record is produced by the vector multiplication of **a** by the *i*th column of **P**. The entire record corresponds to the iteration of this multiplication over all the data symbols.

Matrix **G** is generated algorithmically from *k* x (*s*+1) extended Vandermond matrix **V** through appropriate elementary row transformations [LS99]. Different values of *k* lead to different elements in **P**, despite same *s*. In practice, there are only *n* columns **p** of **P** present in the file. Each parity bucket contains only the corresponding **p**. It suffices as it was shown for the vector multiplication **ap**.

Example

We continue with *GF*(16). The maximal segment size is *s* = 17. We assume the group size *k* = 4. Our group availability level can then scale potentially to 13-availability.

The matrix **G** is:

1	0	0	0	8	F	1	7	7	9	3	C	2	A	E	7	7
0	1	0	0	F	8	7	1	9	7	C	3	A	2	7	E	7
0	0	1	0	1	7	8	F	3	C	7	9	E	7	2	A	7
0	0	0	1	7	1	F	8	C	3	9	7	7	E	A	2	7

It has 4x4 matrix **I** at the left side, all other columns form **P**. Assume the record group with four data records having respectively the non-key data as follows:

En arche ..., Dans le ..., Am Anfang ..., In the Beginning…

The respective symbols in the ASCII encoding are in hexadecimal notation: “45 6E 20 41 72 …”, “41 6D 20 41 6E …”, “44 61 6E 73 20 …”, “49 6E 20 70 74…”. The first symbols form vector **a** = (4,4,4,4). The codeword is **u** = **aG =** (4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,0). Next vector **a** is (5,1,4,9) leading to **u** = (5,1,4,9,F,8,A,4,B,1,1,2,7,E,9,9,A). Here, in particular, the vector multiplication of **a** and for instance of 5th column of **G** leads to F since:

$$\mathbf{a} \cdot (8,F,1,7) = (5,1,4,9) \cdot (8,1,F,7) = 5{\cdot}8 + 1{\cdot}F + 4{\cdot}1 + 9{\cdot}7 = E+F+4+A = F .$$

Notice that all the scalar operations are in the $GF(2^4)$ field. Note also that the first four coordinates of **u** always replicate **a.** Continuing in this manner, we obtain “4F 63 6E E4 …” for the first parity record, “48 6E DC EE …” for the second one, “4A 66 49 DD …” for the third etc.

In the actual file, a copy of 1st column of **P** resides in the header of 1st parity bucket of every record group. Likewise, any *i*th parity bucket has only the *i*th column.

3.2.3 Actual Parity Encoding using RS-codes

The above schema was presented as if all the records in the group entered the file simultaneously. Actually, the parity calculus is performed for each file modification, i.e., an insert, an update and a deletion. Each insert is formally dealt with as vector **a** with all but one coordinates equal to 0. The result **aG** is added to the current **u**, i.e. **u := u** + **aG**. This corresponds to the fractioning calculus of the basic schema one row at the time. Assuming that our example records are inserted in left-to- right order, we first multiply successively all the symbols of 1st record by 1st row of **G**, and then we continue with all the symbols of 2nd record etc. The result is the same as for the basic schema, regardless of the order of inserts.

A deletion of a record inserted using vector **a** is formally an insert of vector –**a** whose unique non zero symbol is *-a*. Since both addition and subtraction in a *GF* boil down to XOR, the parity calculus is the same as for an insert of vector **a**. Finally, an update corresponds to the deletion of previous vector **a** and insert of new one **a'**. The calculus u **:= u** + (**a-a')G** realizes it. It means that the data bucket produces a δ-record whose each symbol is the corresponding *a – a'*. The new codeword is computed from the δ-record.

Incidentally, the updates to records managed by LH*$_{sa}$ using the addition in *GF* only are performed also through the δ-record. Operationally, for all the schemes, the parity encoding calculus is performed at each parity bucket, using the column of **G** residing there for the RS based schemes. The data bucket simply sends the copy of the inserted or of the deleted record, or the δ-record to all the buckets where the parity records of the group are.

3.2.4 Record Recovery Schema using RS-codes

Assume now that in a record group with *n* parity records, at most *n* primary or parity records are unavailable. One or perhaps all of the unavailable data records should be delivered nevertheless to the application. To recover these records, one first collects any *k* available data or parity records of the segment. The rows of **G** corresponding to the available records are concatenated to form *k* x *k parity* matrix **H**. **H** is invertible, **G** being derived from **V** using the elementary row transformations only. The inverse matrix $\mathbf{H}^{-1}$ can be computed, e.g., using the well-known Gaussian elimination. One then collects the symbols with the same offset from the *k* records in a vector **b**. By definition, **a**·**H** = **b**, hence **b**·$\mathbf{H}^{-1}$ = **a**. We find the symbols from the missing records among the coordinates of **a**. We now multiply H^{-1} from the left to recover the symbols from the missing records. By running through the entire record we obtain **b** the missing data records. We only have to calculate the inverse matrix once.

Example

We continue previous example, assuming the availability of only the fourth primary record and of the 1st, 2nd and 3rd parity record. Hence, three data buckets are unavailable. The matrix **H** contains the columns 3rd to 6th from **G**:

0	8	F	1
0	F	8	7
0	1	7	8
1	7	1	F

One can show that $\mathbf{H}^{-1}$ is then:

B	F	A	1
C	4	2	0
4	7	D	0
2	D	4	0

To reconstruct the first symbols of the primary records, we form vector **b** = (4,4,4,4) from the first symbols of the remaining four records. Therefore, **b**·$\mathbf{H}^{-1}$ = (4,4,4,4). For the next symbols to recover, one forms **b** = (9,F,8,A) which leads to **b**· $\mathbf{H}^{1}$= (5,1,4,9). The 1st coordinates of the **b** vectors give us the first missing data record 45..., the 2nd coordinates provide the second record 41..., and the 3rd ones recover the 3rd data record 44…. The 4th coordinates are not of practical interest here as they reproduce the 3rd primary record 49....

3.2.5 Bucket Recovery Schemes

Record recovery basically delivers an individual record to the application. Bucket recovery recovers all the records in the unavailable bucket and stores them elsewhere in the file. As bucket recovery is a much longer operation, an individual record recovery can be performed concurrently.

Every discussed schema shares the principle that the recovered buckets are recreated at spare servers. In other words, there is no redistribution of recovered records among existing buckets, unlike in some well-known schemes. Hence, an *n*-available file should basically dispose of *n* spare servers. IAMs make aware the clients of new locations of recovered buckets.

Bucket recovery for the discussed schemes basically loops over the individual record recovery. An additional part consists of the schema specific determination of records that became unavailable. Next, the calculus is made parallel to distribute the load uniformly over the servers of data buckets of the group. Finally, one factors out the steps common to each record recovery, performed then only once. For the schemes using RS-coding, this is especially the case of $\mathbf{H}^{-1}$ calculus.

4 Parity Management

4.1 Scalable Availability

For all the schemes, the file is basically created as data bucket 0. Also the value of *k* is chosen to be a power of 2. Every record group is provided with one parity record. When the data file reaches *k* buckets, every group starts getting two parity records, to become 2-available. This process occurs incrementally at each split. On the one hand, a new parity record is added to every existing group. For $LH*_{SA}$ and $LH*_{RS}$ these are groups that involve the records that stay in the bucket. For $LH*_g$, this is also the case of all the records that move. On the other hand, any new group gets from the beginning two parity records. For $LH*_{SA}$ and $LH*_{RS}$, any record that the split moves initiates a new group.

When an LH* file has 2^l buckets, next bucket to split is always bucket 0. After *k* more splits, every record group in the file has two parity records. From now on the file is 2-available. This lasts till it scales to k^2 buckets. Then, record groups start getting three parity records. When the file doubles again, all record groups are 3-available. When the file reaches k^4 buckets groups with four parity records start etc.

Increasing *n*-availability incrementally when the file scales up are necessary to keep the reliability above a constant [BM92]. High-availability schemes with this capability were called *scalable* availability schemes. Among the schemes using the parity data, the discussed schemes are the only scalable availability schemes known.

The policy of increasing the *n*-availability level when the file reaches some fixed size is called *with uncontrolled* reliability. Performance analysis for various values of *k* and divers probabilities *p* of bucket failure show that this policy may not suffice. For a larger *k* or smaller *p* the reliability may start dropping for a larger file. For *p* close to 1 or small *k*, it may lead to too many parity buckets for a sufficient reliability, uselessly deteriorating the file access and storage performance. In these cases one should start earlier or delay the increase to *n* with respect to the uncontrolled schema. This capability is called *reliability control*.

All discussed schemes allow for the reliability control. Given *k* and *p*, the file coordinator performs the reliability calculus whenever data bucket 0 should split. The calculus estimates whether the reliability will remain above given threshold *T* until bucket 0 splits again, i.e., until the data file doubles. The creation of additional parity records is started or not accordingly. For $LH*_{SA}$ the control may also incrementally adjust the value of *k* doubling it or decreasing by half. This frees the user from the corresponding choice that could prove sub-optimal. The analysis shows that the reliability control with or without variable *k* may suffice in cases when the uncontrolled scalable availability does not.

4.2 Parity Files

Parity records are stored in parity buckets, separate from data buckets. Each bucket has basically the same capacity *b* as the data bucket. One way to group these buckets into files, is to consider that *i*-th parity records for a group form a distinct file F_i. File F_1 is the 1st to be created, followed later on by F_2 and so on. Alternatively, one may view all F_i's as a single parity file. The choice is merely the implementation convenience. For what follows we stick to the 1st one, easier for the didactic purpose.

For $LH*_g$, each F_i is an LH* file where the parity records are identified by their keys (*g*,*r*). For the other schemes, basically, bucket *g* of file F_i stores all and only i^{th} parity records with different ranks *r* but sharing *g*, called then *bucket group number*. Hence, assuming for instance $LH*_{RS}$ and $k = 4$ again, bucket 0 of each F_i has all the i^{th} parity records of the record groups whose data records are in buckets (0,1,2,3). Likewise, buckets 1 contain respectively the i^{th} parity records for data buckets (4,5,6,7) etc.

The single parity bucket *g* supports the processing load from data modifications *k* times greater than each data bucket in group *g*. If as often, these operations constitute only a fraction of searches, the parity buckets load is not a practical problem. Otherwise, each bucket *g* can be broke into a multibucket subfile of smaller buckets with capacity of *b*/*k*. This file can be an LH* file, where new buckets are created while the group is built. The parity buckets support then about the same processing load as the data buckets.

The basic assumption for all the schemes is that every server site carries a single bucket. However, the correctness of the schemes require only that each parity record of same record group is in a different bucket, and that none of these records is at the server of a data record of the group. It is possible to share the storage space of a server between a data record and a parity record, provided they are from different groups. This can be attractive if the number of storage servers for the file is constrained.

There are variants of LH*$_{SA}$ and of LH*$_{RS}$ for such sharing. One choice for LH*$_{RS}$ is that ith parity bucket for group m is at ith data bucket of group (m+1). This guarantees that every parity record of the same record group is at different server. If one uses multibuckets, assume that a multibucket has at k buckets, i.e., that $n \leq k$ in practice, numbered internally 0,1…k-1. Each multibucket starts then at 1st server of group (m+1). The ith multibucket uses to distribute the parity records among its buckets, the hash function $h_i = h_{i-1} + 1 \bmod k$. Each parity record of a record group is again at a different server, and all the parity records of group m are at the servers of group (m+1). The whole approach reduces the number of additional servers for the file to k at most, i.e., about M/k times. Every server supports a data and a parity bucket, except for servers of group 0 that have only data buckets, and the last servers that support only the parity buckets of last group, let it be m'. If the file expands further, these servers will get data buckets of group (m'+1). New servers will be appended to store at first the parity buckets of that group, and later on also the data buckets of next group etc.

Provided there are at least group 0 and group 1 in the file, each parity bucket of the last current group may be finally temporarily put at the data bucket with the same offset within group 0. If further splits occur, these parity buckets are progressively swapped back to the locations they would have otherwise, i.e. towards the servers of data buckets of group (m'+1). Parity buckets of that group take their place one by one instead. This variant minimizes the number of supporting servers to only the servers of the data buckets. The servers of the basic LH* schema without the high-availability suffice, regardless of n in practice. This is potentially highly attractive, and has the nice theoretical flavor. The storage overhead is n or n+1 parity buckets of capacity of b/k per server of an n-available file. If a group size is for instance k = 32, the storage overhead for a 3-available file is about 10%. This seems an acceptable price for many applications. The main drawback with respect to the previous schemes, is higher split cost due to the swapping.

One can observe finally that n-available file is provided with parity files F_i up to $i = n$ or $i = n + 1$ depending on the file state, for all the discussed schemes and regardless of the parity file structure. With respect to the internal structure of the parity buckets, those of LH*$_{RS}$ and LH*$_g$, contain each "its" column of **G**. Other aspects are implementation dependent.

To send out an inserted or deleted record or a δ-record, the data buckets use the addressing data in its header. These data are updated during the split and through IAMs. Addressing errors may occur because of the splits in an LH*$_g$ parity file or since a parity bucket moved to a spare. Details are in the papers about each schema.

4.3 Availability Management

A client or a server can detect unavailability. In all the cases, the problem is forwarded to the coordinator. In the case of a search or an insert, the coordinator may attempt the delivery, since the unavailability could occur on an incorrect bucket. Otherwise, for a search, it performs the record recovery. In all the cases, it manages the bucket recovery.

In addition to the LH* file-state data, the coordinator has file-state data specific to the availability. These include the file availability level n, record group size k, data for the availability control… For the high-availability of file-state data themselves, given their small size and infrequent updates, they are trivially replicated on n buckets.

5 Comparative analysis

While all the discussed schemes satisfy the basic "buzzword" goals, their differences make them more or less fitting particular application needs. Main mutual advantage and, in turn, drawbacks are as follows. Precise analysis is an on-going research goal.

- LH*$_{RS}$ may probably offer the smallest storage overhead. This is a remote consequence of the property of RS-codes to be MDS (maximal distance separating). The number of parity records to make a group n-available is exactly n which is the theoretical minimum. The number of parity records per the same LH* data file made n-available, may be substantially lower than for LH*$_{SA}$. The total storage space for the parity records may be probably lower than for LH*$_g$, although LH*$_g$ requires the same number n of parity records per n-available group. LH*$_{RS}$ record recovery cost should typically be lower than that of LH*$_g$, but may be substantially higher or substantially lower than that of LH*$_{SA}$. This is due to more complex parity calculus of LH*$_{RS}$ on the one hand, or to possibly more messages for LH*$_{SA}$ to explore multiple groups in the case of a multiple failure, on the other hand.

- LH*$_g$ has the smallest split cost. Its groups are location independent, hence splits do not need to recalculate existing parity data. As a data bucket may contain thousands of records, access performance of LH*$_g$ may be substantially better, especially for highly scaling files. In contrast, its record recovery cost should typically be the highest, at least for a single unavailability. The reason is that the parity record can be in any bucket of its F_1. Since its key is unknown, it requires a parallel scan initiated through multiple unicast messages or through a multicast. In the other schemes the bucket is directly known, and a single unicast message suffices.
- Finally, LH*$_{SA}$ may attract an application because of the relative advantages already discussed. Besides, it often has the capability to recover from of $m > n$ unavailabilities. The difference may be quite substantial. For instance, a 2-available LH*$_{SA}$ file may recover records from any k unavailable buckets in the same group. This is impossible for LH*$_{RS}$ that can accommodate at most 2 unavailable buckets per group in a 2-available file. On the other hand, LH*$_{RS}$ has its own “good cases” that LH*$_{SA}$ can’t handle. For instance a “square” of 4 buckets 0,1,4,5 in our example file with $k = 4$ for LH*$_{SA}$, assumed 2-available. The overall balance seems nevertheless largely in favor of LH*$_{SA}$.

6 Conclusion

Record grouping provides attractive properties, including the scalable availability. We have shown that schemes with different characteristics can be built around the same basic concepts. We have compared three LH* based schemes. Each leads further to many variants tuning selective features. These are not dealt with in our comparative analysis. Different performance of various schemes broaden the class of applications that might benefit from.

Among these are modern database systems, more and more often needing large scalable databases, and for which the parallel access is already a must [FBW97], [B&al95]. Many of these databases grow very rapidly. The already mentioned UPS database passed from 4 to 13 TB between 97 and 98, and many other similar examples are known. The multimedia servers also start using multicomputers and the success may make them scaling big [B&al95], [H96]. In the Web arena, more and more systems maintain TB of data on large multicomputers. The popular Inktomi systems, [I98], typically turn over dozens of interconnected computers, as their flagship configurations already cited in the Introduction. An implementation of SDDSs is already under study for such systems [G99]. For all these applications both scalability and 24/7 availability are critical. The already mentioned mishap of Ebay is here to stay to remind.

Future work should include more in-depth comparative analysis of the discussed schemes. This should be done through theoretical studies and through implementations. The basic ideas should be applied to other known SDDSs to create their high-availability variants. The variants of discussed schemes should also be studied. The variant of LH*$_{RS}$ whose parity records are at the sites of data records only, seems immediately applicable to the current generation of parallel database systems using static hash or range partitioning. Finally, Reed-Salomon codes are one attempt to achieve the scalable availability. Other choices seem possible, [ABC97], [BBM93], or [H&a94].

Acknowledgments *This research was partly sponsored by Research Grant of IBM Almaden Res. Cntr. to Centre d’Etudes et de Recherches en Informatique Appliquée (CERIA) d’Université Paris 9 Dauphine.*

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.

[B99] Bertino & al. Indexing Techn. for Advanced Database Systems. Kluver, 1999

[B99a] Bartalos, G. Internet: D-Day at eBay. Yahoo INDIVIDUAL INVESTOR ONLINE, (Jul 19, 1999).

[B&al] Boloski & al. The Tiger Video Server. http://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.

[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.

[G99] Gribble, S. Cluster-Based Internet Services with SDDS. Master Th. UC Berkeley, 1999.

[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.

[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).

[LC93] Lin, S., Costello, D. J. .Error Control Coding: Fundamentals and Application, Prentice-Hall, 1993.

[LMR98] Litwin, W., Menon J., Risch, T..LH* with Scalable Availability. With IBM Almaden Res. Rep. RJ 10121 (91937), (May 1998).

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

[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.

[LS99] Litwin, W., Schwarz, J.E.. LH*$_{RS}$: A High-Availability Scalable Distributed Data Structure using Reed Solomon Codes. CERIA Res. Rep. 99-2, U. Paris 9, 1999.

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

[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.

[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.

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

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

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

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

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