LH*$_{RS}$: A High-Availability Scalable Distributed Data Structure using Reed Solomon Codes

Witold Litwin
Université Paris 9 (Dauphine),
Pl. du Mal de Lattre, Paris 75016, France,
Witold.Litwin@dauphine.fr

Thomas Schwarz, S.J.
Jesuit School of Theology
1756 Leroy Avenue, Berkeley, CA 94709, USA,
schwarz@scudc.scu.edu

ABSTRACT

LH*$_{RS}$ is a new high-availability Scalable Distributed Data Structure (SDDS). The data storage scheme and the search performance of LH*$_{RS}$ are basically these of LH*. LH*$_{RS}$ manages in addition the parity information to tolerate the unavailability of $k \geq 1$ server sites. The value of k scales with the file, to prevent the reliability decline. The parity calculus uses the Reed-Solomon Codes. The storage and access performance overheads to provide the high-availability are about the smallest possible. The scheme should prove attractive to data-intensive applications.

Keywords

SDDS, scalable, high-availability, Reed-Solomon Codes

1 INTRODUCTION

Multicomputers (collections of computers connected by a high-speed network) are claimed to be the industry choice for the next millennium [M97], [P98]. They combine affordability and high performance, but also demand new data structures and algorithms, [CACM97]. Specifically, the need for scalability led to the proposal of Scalable Distributed Data Structures (SDDS) [LNS96]. SDDSs allow for files whose records reside in buckets at different server sites. The files support key-based searches and parallel/distributed scans with function (query) shipping. They can be hash-partitioned, or ordered with respect to the primary key or support multikey access. Among the SDDS studied [SDDS], probably the best known is the distributed version of Linear Hashing [L80], called LH*, [LNS96], [KLR96], [B99a], [K98], [R98], [SDDS].

An SDDS file is manipulated by the SDDS client sites. Each client has its own addressing schema called image that it uses to access the correct server where the record should be. As the existing buckets fill up, the SDDS splits them into new buckets. The clients are not made aware synchronously of the splits. A client may have an outdated image and address an incorrect server. An SDDS server has the built-in capability to forward incorrect queries. The correct server sends finally the Image Adjustment Message (IAM) to the client. The information in an IAM avoids at least repeating the same error twice. It does not necessarily make the image totally accurate.

MOD 2000, Dallas, TX USA

These principles avoid the centralized address calculus that could become a hot spot. They allow SDDS files to scale to thousands sites. The scaling makes however a bucket unavailability (failure) increasingly likely. A high availability scheme retains the accessibility of all records to the application despite failures. A k-availability scheme preserves the availability of all records despite up to k bucket failures. A 0-availability scheme does not tolerate any unavailability. The LH* and the traditional data structures are 0-available by this measure.

Higher values of k enhance the reliability of the file, i.e., the probability that all stored data are available to the application, [LMR98]. Modern databases run the 24/7 regime under web access and exemplify the need for high availability schemes as well as the cost of data unavailability. The well-known crash of eBay in June 1999 resulted in the loss of $4B of market value and of $25M in operations [B99]. The failure of a typical financial database costs $10K-$27K per minute.

The first 1-available SDDS scheme was a variant of LH* called LH*$_M$, using record mirroring [LN96]. The scalable and distributed generalizations of B+-trees, [BV98] and [VBW98], also use the replication. In both cases, the doubling of the storage cost may be prohibitive. High-availability variants of LH* with lower storage overhead have therefore been developed. The 1-availability scheme LH*$_S$ stripes every data record into m stripes, then places each stripe into a different bucket and stores the bitwise parity of the stripes in parity records in additional parity buckets, [L&al97]. The storage overhead for the high-availability is only about 1/m for m stripes per record.

Striping produces typically meaningless record fragments, and usually impairs the parallel scans requiring entire records at each site. Efficient scans are decisive for many applications of web servers or parallel databases, [R98]. Another 1-availability variant of LH* termed LH*$_g$ addressed this concern [LR97], [L97], [LMRS99]. The application records, called *data records*, remain entire in LH*$_g$. For the high-availability, they form *m*-member *record groups* provided each with the bitwise parity record. The storage overhead is about 1/*m*, as for the striping. The speed of searches and of parallel scans without failures is that of generic (0-available) LH*. It is unaffected by the additional structure for the high-availability.

The 1-availability or even *k*-availability for any static *k* cannot prevent the reliability decrease in a scaling file. The *scalable availability scheme* LH*$_{SA}$ was therefore designed to dynamically increase *k*, [LMR98]. It uses an elaborated record grouping, where each data record *c* is a member of *k* or *k*+1 groups that intersect only in *c* and are 1-available. For each *k*, the LH*$_{SA}$ file is *k*-available. The storage overhead may vary substantially while the file scales. It can be close to *k*/*m*, the known minimum for *k*-availability, [H&al94]. But, it can also reach over 50 %.

Below, we present an alternative scalable availability scheme termed LH*$_{RS}$. Through record grouping, it retains the LH* generic efficiency of searches and scans. The *k*-availability results from *k* or *k*+1 (generalized) parity records per record group. The parity calculus uses the Reed Solomon (RS) Codes. This mathematically complex tool proves simple and efficient in practice. The advantages of LH*$_{RS}$ are "smoother" storage overhead, always close to the minimal possible, and faster recovery algorithm. The capabilities of LH*$_{RS}$ make the scheme attractive. The other high-availability LH* scheme retain nevertheless some advantages. The diversity should prove attractive to implementers.

Section 2 introduces the LH*$_{RS}$ scheme, focusing on its use of RS calculus. Section 3 presents the actual parity computations in an LH*$_{RS}$ file. We explain the file manipulation in Section 4. Section 5 discusses file performance and Section 6 addresses variants to the scheme. Section 7 presents related work, and Section 8 concludes the article. The Appendix provides some mathematical background of RS-codes and pseudo-code for some algorithms.

2 HIGH-AVAILABILITY OF LH*$_{RS}$ SCHEMA

2.1 Record Grouping

We assume the basic familiarity with LH*. A LH*$_{RS}$ file consists of an LH*-file called *data file* with the *data records* generated by the application in *data* buckets 0,1,…,*M*-1. The LH* data file is augmented for the high-availability with *parity* buckets to store the parity records at separate servers. A data record is identified by its key *c* and has also some non-key part. As for the generic LH*, the correct bucket *a* for data record *c* in an LH*$_{RS}$ file is given by the *linear hashing* function $h_{j,n}$; $a = h_{j,n}(c)$. The parameters (*j*,*n*) called *file state* evolve dynamically. The client image consists also from *h*, but perhaps with outdated state. Details of the address computations are not important here.

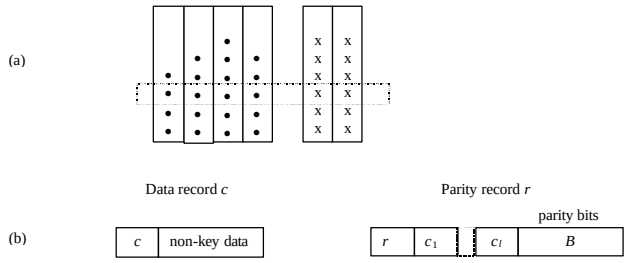

Figure 1. LH*$_{RS}$ file: (a) 2-available bucket group, (b) Data and parity records.

We group successively created data buckets in *bucket groups*. All but perhaps the last bucket group have the same size $m = 2^f$ for some $f > 1$. Bucket *a* belongs to the group numbered $g = a$ **div** *m*, where **div** denotes integer division. The last bucket group can contain less than *m* buckets. For the parity calculus, we formally complement it with dummy (not really existing) buckets with dummy (zero) records. Every bucket group is provided with $k \geq 1$ parity buckets where the parity records for the group are stored. Figure 1a shows a bucket group with four data buckets and their data records (•) and two parity buckets and their parity records (x). Each data record has a *rank* 1,2… that reflects the position of the record in its data bucket. A *record group* contains all the data records with the same rank *r* in the same bucket group. The record group with $r = 3$ is enclosed for example in Figure 1a. The *k* parity buckets contain parity records for each record group. A parity record consists first of the rank *r* of the record group, then of the primary keys c_1, c_2, ¼ c_l of all the (non-dummy) data records in the record group, and finally of the (generalized) *parity data* calculated from the data records and the number of the parity bucket. The parity data, denoted by *B* in Figure 1b, differ among the parity records. We call the ensemble of the data records in a record group and its parity records a *record segment* and likewise, the bucket group with its parity buckets a *bucket segment*.

2.2 Scalable Availability

Figure 2 illustrates the expansion of an LH*$_{RS}$ file. Parity buckets are represented shaded and above the data file, right to the last, actual or dummy, bucket of the group they serve. For the sake of the example, we choose $m = 4$. Dummy buckets are delimited with dotted lines. The file is created with data bucket 0 and one parity bucket, Figure 2a. The first insert creates the first parity record, calculated from the data record and from *m*-1 dummy records. The data record and the parity record receive rank 1. The file is 1-available.

When new records are inserted into data bucket 0, new parity records are generated. Each new data and parity record gets the next rank. When data bucket 0 reaches its capacity of *b* records, $b >> 1$, it splits. Usually half of its records move into data bucket 1. During the split, both the remaining and the relocated records receive new consecutive ranks starting with $r = 1$. Since there are now (non-dummy) records with the same rank in both buckets, the parity records are recalculated, Figure 2b.

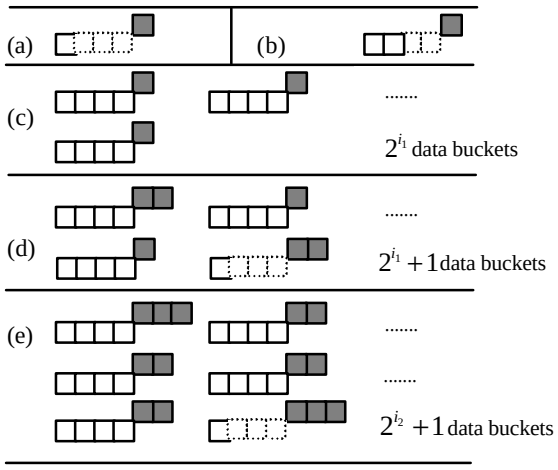

Figure 2. Scalable availability of LH*$_{RS}$ file. (a) initial file, (b) first split, (c) max. 1-available file, (d) beginning 2-availability and (e) 3-availability.

The continuing growth of the file through inserts formally replaces the dummy records in the data buckets with actual records. The splits append new data buckets 2,3… Eventually, a split creates data bucket *m*+1. This starts the second bucket group and its first parity bucket.

While the file continues to scale, the probability of double failure increases. To offset the decline in reliability, the file availability increases gracefully to two parity buckets per group. This process

starts, Figure 2c and Figure 2d, when the file reaches some size $M = 2^{i_1}$. Next bucket split is then bucket 0. We recall that LH* scheme splits the buckets in the deterministic order $0,0,1\ldots 2^i - 1,0\ldots$ On the one hand, each new bucket group is from now on formed with two parity buckets. On the other hand, a second parity bucket is appended to each existing bucket group during the split of its 1st member. When M reaches $M = 2^{i_1+1}$, i.e., when the file has doubled in size, all groups have two parity buckets. At this point, the file is 2-available and can survive failure of any two buckets.

Further scaling makes a triple failure increasingly likely. To offset the reliability decline, one has to further increase the availability level. This starts again when the file reaches some size $M = 2^{i_2}$; $i_2 > i_1$ so that next bucket to split is again bucket 0, Figure 2e. Each split, starting from that of bucket 0 again, creates then three parity buckets for the newly appended data bucket, and adds a third parity bucket to the two already present for the splitting data bucket. Once data bucket $M = 2^{i_2}$ splits in this way, all bucket groups are 3-available, and the file has reached 3-availability. This process can continue towards 4-availability, etc., making LH*$_{RS}$ a *scalable availability* schema.

The values of i_1, i_2 etc. which determine when the availability level starts to increase, are controlled by the LH*$_{RS}$-*coordinator*. Essentially, the LH*$_{RS}$ coordinator is the LH* coordinator provided with additional capabilities for the high-availability. We recall that the LH* coordinator handles the file state parameters which it uses to calculate next bucket to split when some, usually another, bucket reports an insert creating an overload. In addition, the LH*$_{RS}$ coordinator initiates the creation of parity buckets for new groups and the scaling up of the availability. It also manages the record and bucket recovery.

The scheme allows for a variety of strategies in the management of availability. The basic strategy starts increasing the k-availability towards $(k + 1)$-availability whenever the files reaches m^k buckets. In the notation from above, this variant chooses $2^{i_1} = m$, $2^{i_2} = m^2$ etc. We call this strategy, as any other using predefined values of i, *uncontrolled reliability*. We implement the *reliability control* strategy by dynamically choosing i based on the file reliability as monitored by the coordinator. Whenever bucket 0 is next in line for a split, the scheme decides whether the reliability level would drop under some threshold P_{min} before bucket 0 is again about to be split, and starts to create $(k + 1)$ parity buckets if necessary. The basic P_{min} is the reliability of a single bucket. Section 5.4 discusses the reliability control more in depth.

2.3 Parity Coding

LH*$_{RS}$ parity calculus uses the linear RS codes. These are originally error correcting codes, among most efficient, since they are maximal distance separating (MDS) codes. We use them as *erasure correcting* codes recovering unknown data values. We first recall the theory of Galois Fields, at the basis of the RS codes. We use terms from [MS97].

2.3.1 Galois Fields

A Galois Field $GF(N)$ is a set with N elements and the arithmetic operations of addition, subtraction, multiplication, and division. There are two distinguished elements, a zero element, written 0, and a one element, written 1. The operations over $GF(N)$ possess the usual properties of their analogues in the real numbers including the properties of 0 and 1.

For LH*$_{RS}$, we only use $GF(2^f)$ for some $f > 0$. We represents the elements of the field as f-bit strings. The byte based structure of modern computers suggests $f = 4$ or $f = 8$. Generally, the implementation of multiplication consumes more resources for larger values for f, whereas a smaller value of f limits too much the number of parity records in a record group.

For every f, the bit representation of zero is $0 = 00\ldots0$ and of one is $1 = 0\ldots01$. For bytes, one thus has 0 = 0000 0000 and 1 = 0000 0001. The addition and the subtraction of two elements are the same and equal to their Exclusive-Or (XOR). The definition of multiplication and division is more cumbersome. Mathematically most convenient is the definition of the multiplication based on representing the elements of $GF(2^f)$ as polynomials of degree f over the field $GF(2) = \{0,1\}$. The multiplication is then the multiplication of polynomials and the product is reduced modulo a certain generator polynomial. These generator polynomials are irreducible polynomials of the appropriate degree. The mathematical tables with generator polynomials for interesting field sizes are in [MS97].

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 the multiplication in *GF*(16).

GF multiplication via polynomial multiplication is hardly efficient. Rather, one uses table look-up. Two methods are attractive: (1) a complete multiplication table supplemented by a table of multiplicative inverses, and (2) the logarithm and the antilogarithm tables. Method (1) is conceptually the easiest. Its drawback is the size of the multiplication table. For bytes, the multiplication table would contain $2^8 \cdot 2^8 = 2^{16}$ entries or 64KB. In addition, the table is two-dimensional and calculating the address of an entry introduces additional overhead.

Method (2) is based on the existence of the *primitive* elements in any *GF*. The property of each primitive element p is that every non-zero field element is a power of p. We choose p and determine for each element g in the *GF* the power i such that $p^i = g$. We call i the *logarithm* of g and write $i = log_p(g)$. Inversely, we call g the *anti-logarithm* of i and write $g = antilog_p(i)$. We tabulate logarithms and antilogarithms in two tables, each of the size of the field. We thus have 2^{f+1} entries in total, many times less than method (1) requires for a larger field, e.g. only 512

entries for $GF(256)$. We implement multiplication and division of Galois field elements using these tables. For every two elements g and $h \in GF(2^f)$:

$$g \times h = antilog_p (log_p(g) + log_p(h))$$
$$g/h = antilog_p (log_p(g) - log_p(h)).$$

The addition or subtraction is here modulo $2^f - 1$ which is the number of non-zero elements in GF.

2.3.2 Example

Consider $GF(16)$ so $f = 4$ and there are 15 non-zero elements. There are different implementations of this field, one is in Table 1. Each field element can be represented as a bit string, an integer, and as a hexadecimal digit. The primitive element is 2, the zero element is 0, and the one element is 1.

Addition remains the XOR operation as given by the ^ operator in C, C++, and Java. As an example of arithmetic, we calculate the sum, product, and quotient of the two field elements $A = 10$ and $B = 11$ in the integer notation. For the sum, we calculate $A+B = 1010 \wedge 1011 = 0001 = 1$. For the product, we take the logarithms 9 and 7, add them up modulo 15, to obtain 1. The number with logarithm 1 is 2, hence A*B = 2. To calculate A/B, we subtract the two logarithms, by taking the remainder modulo 15 we change the difference to a number between 0 and 15, and then we take the antilogarithm. Since $log(A) - log(B) = -2 \equiv 13$, and since $antilog(13) = D$, we have A/B=D. In the faster version of method (2), we use the offset –2 into the extended anti-logarithm table.

2.3.3 Parity Encoding

We use n for the maximal segment size, m for the record group size, and k for the number of parity buckets. Thus, $n = m + k$, and the group is k-available.

For the parity calculus, we identify the data record with its non-key field and the parity record with its parity field B. Since the data record key is replicated in the parity records, it becomes part of the parity calculus. We assume that data records in a record group are of the same length. Otherwise we pad the shorter records with 0 bits to obtain the same length. We treat any record as a bit string. We break each string into *symbols* of length f. If f does not divide the length of the string, we again pad with 0 bits. If we choose $f = 4, 8$, this padding should not be necessary. We identify the set of all possible symbols with the elements in $GF(2^f)$.

We generate each parity record for the record group from the data records one symbol at a time. For the sake of presentation, we assume that all parity records are generated at the same time. Similarly, we assume that all data records in the record group are inserted into the file simultaneously. We show the actual operations in Section 3. Finally, we assume the coding calculus to be centralized, while – as we will see - it is distributed in the LH*$_{RS}$ scheme.

We are thus given m data records, each of which is a string of symbols. We calculate now the first symbol in all the parity records simultaneously. Subsequent symbols are calculated in precisely the same manner. First, we form the vector $\mathbf{a} = (a_1,a_2,a_3,...,a_m)$ where a_i is the symbol from the i^{th} record in the group. We collect the first symbol in all records (data and parity) in the vector $\mathbf{u} = (a_1,a_2,...,a_m,a_{m+1},...,a_n)$, to which we refer as a *code word*. The first m coordinates of **u** are the coordinates of **a.** The remaining k coordinates of **u** are the newly generated parity bucket symbols.

We obtain **u** from **a** by multiplying **a** from the right with a *generator matrix* **G** of the linear (systematic) RS-code; namely **u** = **a G.** **G** has m rows and n columns. **G** is *systematic*, that is, **G** consists of two concatenated sub matrices; namely **G** = **I**|**P**. Matrix **I** is a m x m identity matrix, hence **a I = a**. That is why first m coordinates of **u** form **a**. Only the columns of the *parity* matrix **P** operationally contribute to the k 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 through appropriate elementary row transformations from m x n matrix **V** that is a Vandermonde matrix (either simple or extended) [MS97]. The algorithm is in Appendix A. Different values of k lead to different elements in **P**, despite same n. In practice, there are only the k columns **p** of **P** present in the file. Each parity bucket contains only one **p**. This suffices as the parity symbol is the result of the product of **u** with the column **p**.

The maximum number of columns of **G** is $2^f + 1$, e.g. 257 for byte sized symbols. The number of parity records for a record group is limited by this bound, which however appears to be sufficient for byte sized symbols. We know a way to overcome this bound dynamically, but this method is beyond our scope here.

2.3.4 Example

For the sake of simplicity, we continue with $GF(16)$. The maximal segment size supported by $GF(16)$ is $n = 17$. We set the bucket group size to $m = 4$. Our file availability level can scale to 13-availability. There is a way to allow even higher availability through dynamic switch to the field $GF(256)$, although we will not present it. We calculate a generator matrix **G** as in Appendix A to be

$$\mathbf{G} = \begin{pmatrix} 1 & 0 & 0 & 0 & 8 & F & 1 & 7 & 7 & 9 & 3 & C & 2 & A & E & 7 & 7 \\ 0 & 1 & 0 & 0 & F & 8 & 7 & 1 & 9 & 7 & C & 3 & A & 2 & 7 & E & 7 \\ 0 & 0 & 1 & 0 & 1 & 7 & 8 & F & 3 & C & 7 & 9 & E & 7 & 2 & A & 7 \\ 0 & 0 & 0 & 1 & 7 & 1 & F & 8 & C & 3 & 9 & 7 & 7 & E & A & 2 & 7 \end{pmatrix}$$

The left four columns of **G** form the identity matrix. Any four different columns of **G** form an invertible matrix. The multiplication of four-dimensional vector **a** by **G** leads to 17-dimensional vector. Because of the 4 x 4 identity submatrix of **G**, the first four coordinates of the vector replicate **a**. The other 13 coordinates are symbols for successive parity records.

Assume the following four data records: “En arche ...”, “Dans le ...”, “Am Anfang ...”, “In the beginning…”. The bit strings in GF corresponding to the ASCII encoding for our four records are (in hexadecimal notation): “45 6E 20 41 72 …”, “41 6D 20 41 6E …”, “44 61 6E 73 20 …”, “49 6E 20 70 74…”. To calculate the first symbols in each parity record we form the vector **a** = (4,4,4,4) and multiply it by **G.** The product is vector **u** = (4,4,4,4,4,4,4,4,4,4,4,4, 4,4,4,4,0). For the second symbol in each parity bucket, we form the vector **a** = (5,1,4,9) and multiply by **G** to obtain code word **u** = (5,1,4,9,F,8,A,4,B,1,1,2,7,E,9,9,A). Notice that the first four coordinates of **u** always replicate the coordinates of **a.** We do not have to calculate all coordinates of **u** at once, but can calculate them individually instead. For example, to calculate the fifth coordinate of 2nd **u**, we multiply vector **a** with the fifth column of **G**. Expressing this more conveniently using the dot product, we calculate (using GF instead of integer operations):

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

$= E + F + 4 + A = F.$

The matrix notation merely combines all 17 dot product calculations. In this manner, we obtain "4F 63 6E E4 …" for the first parity record, "48 6E DC EE …" for the second parity record, and "4A 66 49 DD …" for the third.

2.3.5 Record Recovery

Assume that LH*$_{RS}$ finds at most k data or parity records of a record segment to be unavailable. Collect any m available records of the segment. Also, concatenate the corresponding columns of **G** into the m x m matrix **H**. By virtue of the Vandermonde matrix, any m x m submatrix of **G** is invertible. Using for example Gaussian elimination, we compute $\mathbf{H}^{-1}$. Collect the symbols with the same offset from the m records into a vector **b**. By definition, $\mathbf{a} \cdot \mathbf{H} = \mathbf{b}$ implying $\mathbf{b} \cdot \mathbf{H}^{-1} = \mathbf{a}$. Hence, multiply **b** by $\mathbf{H}^{-1}$ to recover the missing symbols with the same offset. Using the same $\mathbf{H}^{-1}$, iterate through the entire available records, to produce all missing records.

2.3.6 Example

Consider that first three data records above became unavailable, i.e., only the fourth data record and the first, second and third parity records are available. We form **H** from the columns 3 to 6:

$$\mathbf{H} = \begin{pmatrix} 0 & 8 & F & 1 \\ 0 & F & 8 & 7 \\ 0 & 1 & 7 & 8 \\ 1 & 7 & 1 & F \end{pmatrix}$$

Inversion of **H** yields:

$$\mathbf{H}^{-1} = \begin{pmatrix} B & F & A & 1 \\ C & 4 & 2 & 0 \\ 4 & 7 & D & 0 \\ 2 & D & 4 & 0 \end{pmatrix}.$$

The first vector **b** formed from the first symbols of the three remaining records is **b** = (4,4,4,4). Hence, $\mathbf{b} \times \mathbf{H}^{-1}$ = (4,4,4,4). The next symbols lead to **b** = (9,F,8,A) and $\mathbf{b} \cdot \mathbf{H}^{-1}$= (5,1,4,9) etc. The first coordinates of **b** vectors provide the first missing data record "45...", the second coordinates the second data record "41...", the third coordinates the third data record "44…", and the fourth coordinates merely reproduce the fourth data record "49..."

3 ACTUAL PARITY CODING

The basic operations on a data record are that of insertion, deletion, or update. For the parity calculus, the update is the generic operation, the inserts and deletes are seen as special cases. An update basically changes only the non-key data, and related parity records. An update to the key is dealt with as a deletion followed by an insert into usually a new location. An insert is formally an update of a dummy record into the actual one. Vice versa, a deletion is an update into the dummy record.

3.1 Updates

We consider an update to the i^{th} data record in its group. Let **a** be the vector with symbols with the same offset of the data records in the record group before the update and **a'** the vector formed similarly after the update. Vectors **a'** and **a** differ in the i^{th} position only. Le **u** and **u'** be the resulting code words.

Thus **u = a G**, **u' = a' G** and their difference is $\boldsymbol{\Delta} = \mathbf{u} - \mathbf{u}' = (\mathbf{a} - \mathbf{a}')\,\mathbf{G}$. We have $\boldsymbol{a} - \boldsymbol{a}' = (0,...,0,D_i,0,...,0)$ where Δ_i is the difference between the same offset symbols in the old and in the new record. We recall that, in a *GF*, both the addition and the subtraction are equal to the XOR operation. To calculate $\boldsymbol{\Delta}$, we only need the i^{th} row of **G.** Since $\mathbf{u}' = \mathbf{u} + \boldsymbol{\Delta}$, one can calculate the new parity values by calculating $\boldsymbol{\Delta}$ first and then XOR this to the current **u** value that is the content of B field of each parity record. In other words, with $\mathbf{G}_i$ being the i^{th} row of **G**:

$$\mathbf{u}' = \mathbf{a}' \cdot \mathbf{G} = (\mathbf{a} + (\mathbf{a}' - \mathbf{a}))\,\mathbf{G} = \mathbf{a}\,\mathbf{G} + (\mathbf{a} - \mathbf{a}')\,\mathbf{G} = \mathbf{u} + D_i\,\mathbf{G}_i.$$

In particular, the new symbol u'_j in bucket j is calculated from the old symbol u_j, the difference D_i between the new and the old symbol in the updated record, and the coefficient of **G** located in the i^{th} row and j^{th} column as

$$u'_j = u_j + D_i\, g_{i,j}.$$

We call $\boldsymbol{\Delta}$-record the string obtained as the XOR of the new and the old symbols with the same offset within the updated record.

To implement an update operation without key change, LH*$_{RS}$ sends the $\boldsymbol{\Delta}$-record together with the bucket number i and the rank r (to identify the record group) to all parity sites. Each parity site calculates the B field (or parity record proper) according to equation (3.1). Coefficient g_{ij} is stored with the parity bucket as part of the j^{th} column of **G**.

3.2 Inserts and deletions

An insert formally replaces a dummy record with an actual data record. At the data bucket, the key field c and the non-key data field are updated. Through the insertion, the new record obtains a rank r. The data bucket then sends the rank r, the key c, the bucket number i, and the non-key data as the $\boldsymbol{\Delta}$-record to all parity buckets. The rank identifies the parity record in need of change. If the parity record with this rank does not yet exist, we create it. The field c_i of this parity record changes to c. We calculate field B in the parity record by XORing the $\boldsymbol{\Delta}$-record with the current contents of B, just as for an update. The justification lies in the fact that the implicit dummy record has been changed to the new record.

Likewise, a data record deletion is an update to a dummy record. The operation proceeds by first finding the data record and removing it from the data bucket. Rank r, bucket number i, and the record itself as $\boldsymbol{\Delta}$-record are send to all the parity buckets. Rank r identifies the parity record, the field c_i is set to zero, and field B is XORed with the $\boldsymbol{\Delta}$-record. If the data record was the last in the group, then all key fields c_j in the parity record are now zero. The field B is zero as well. We can then delete the parity record as well.

3.3 Example

Continuing with the running example, assume that we have four data buckets 0…3 with two parity buckets for this group, all buckets being empty. We insert, and update records into successive buckets and at the same rank, disregarding, for the sake of the example, the actual LH* addressing rules. First, we insert record "En arche ..." into data bucket 0. This becomes the $\boldsymbol{\Delta}$-record, since the previous content is a dummy record, so XORing the symbols in the string "En arche ..." with the zero symbols yields of course the string "En arche ...". The $\boldsymbol{\Delta}$-record is then sent to the parity buckets, together with its bucket number 0 and rank 1. In hexadecimal notation, the $\boldsymbol{\Delta}$-record is "45 6E 20 41 72 ...". The first parity bucket is bucket 4, hence it carries $\mathbf{G}_4$. There, the symbols of the $\boldsymbol{\Delta}$-record get multiplied by 8, the first row coefficient in $\mathbf{G}_4$. The result is the string "6E 59 30 68 ..." which becomes the field B of the first parity record with rank 1.

Formally, the string resulting the multiplication is XORed to the old content of *B* to become the actual *B*. As the old *B* happens to contain only zero symbols, there is no need to actually perform the XOR. Similarly, the second parity bucket multiplies the ***D***-record with F, the coefficient in the first row of $\mathbf{G}_5$, yielding "96 45 D0 9F ..." in the parity field *B*.

Now, we insert the second record "Dans le ..." or "41 6D 20 41 6E ..." into bucket 1. This string is sent to both parity buckets as the **D**-record. It multiplies there respectively F and 8, which are the coefficient in the second rows of $\mathbf{G}_4$ and of $\mathbf{G}_5$. The results are "9F 47 D0 9F ..." and "68 53 30 68 ..." respectively. These are XORed to the strings in each *B* that are the above "6E 59 30 68 ..." and "96 45 DO 9F ...". This yields to "F1 1E E0 F7..." and "FE 17 E0 F7 ..." as the contents of the *B*-fields in buckets 4 and 5.

We insert the other two data records in the same manner. The final parity record fields *B* become "4F 63 6E E4 …" and "48 6E DC EE …". Assume now that one changes the 1st data record from "En arche …" to "In initio …". In hexadecimal notation, the change is from "45 6E 20 41 72 …" to "49 6E 20 69 …". We XOR these two strings obtaining the **D**-record "0C 00 00 28…", shipped to the parity buckets. This **D**-record is shipped from bucket 0 to both parity buckets. At the first parity bucket, we first multiply it by the coefficient in the first row of $\mathbf{G}_4$. We obtain "0A 00 00 3C …" We form the XOR with the existing *B* that is "4F 63 6E E4…". The result is "45 63 6E D8 …". The calculation at the second parity bucket proceeds in the same manner, except that we use of the coefficient in the first row of $\mathbf{G}_5$, namely F. The multiplication of the **D**-record by F yields "08 00 00 D1 …". The final XOR to the old content of *B* gives "45 63 6E D8 …" as the new value.

4 FILE MANIPULATION

To create an LH*$_{RS}$ file, the application provides the group size *m*. The *GF*(4) or *GF*(8) are chosen accordingly, and the coordinator computes the generator matrix **G**. It also initializes bucket 0 and the first parity bucket, where it stores the first column of parity matrix **P**. All other file creation operations are as for the generic LH*.

Further manipulation is in *normal* mode if it executes as generic LH* operations, except perhaps for additional operations on the parity buckets, assumed all accessible. An operation enters the *degraded* mode if it cannot access a record in normal mode. This happens for an unavailable or *displaced* bucket. The displaced bucket case occurs when a query is sent to the bucket that in the meantime was recovered. Hence, it is elsewhere at the server that was originally a spare. The originator of the query may be unaware of new location. It gets then the correct address if the query terminates successfully in a dedicated IAM. The address is not necessarily of that spare that could itself become unavailable in the meantime.

The operations in degraded mode are handed to the coordinator. From the file state, it may locate the displaced buckets. If a recovery should occur, the coordinator attempts to recover the entire bucket at a spare, or only the searched record. If there are less than *m* available records for a group, which makes the recovery calculus as defined above impossible, the coordinator enters the *catastrophic mode.* Case specific algorithms are then used. Some records may be unrecoverable, but there may be good cases. We will not discuss this mode in depth. An example of a good case is at the end of Section 7.

We now overview the LH*$_{RS}$ file manipulations focusing on differences to generic LH*. We start with record and bucket recovery. Recall that the number of data buckets in a bucket group is *m* and call the number of parity buckets in a given bucket group *k'*. The number of buckets in a segment of a *k*-available file is $n = m + k'$, where $k'=k$ or $k'=k+1$.

4.1 Record Recovery

To recover a record with key *c*, the coordinator probes the *k'* parity buckets in some order, sending an unicast message with key *c*, the unavailable data bucket address *a*, and the address of the client. If no parity bucket is available, then the failure is catastrophic. Otherwise, the first parity bucket *p* that replies takes the control of the recovery, to avoid turning the coordinator into a bottleneck for recovery. Bucket *p* searches for the parity record with *c* among its keys. The rank *r* of the record cannot be determined from *c*, hence one uses a sequential scan, or the parity bucket maintains a hash table with entries of the form (*c*,*r*). If *c* is not found, then the search for the key *c* is unsuccessful, i.e., the record with key *c* was not in the file. Notice that this constitutes also a successful recovery.

Otherwise, the parity record *r* found contains key *c* only or with $x \leq (k'-1)$ other keys. If record *r* contains only *c*, then all the other records in its record group are dummy. One trivially computes offset *i* of *c* within its group using *a* and creates **H** from all columns of **I**, but column *i*, and from its column of **P**. Afterwards, one performs the recovery calculus using the RS-decoding, as described.

If record *r* contains several keys, the bucket searches for every data record with key $c' \neq c$ in record *r*. If all are found, and $x = (k'-1)$, then the bucket orders them along their offsets, produces **H** from all columns of **I** but *i* and from its own parity column and performs the recovery calculus. If $x < k'$, but all the records are found, then the records at the other offsets are assumed dummy.

If however $l > 0$ records among *x* reveal unavailable, and $l < k'$, then bucket *p* probes other parity buckets. Each probe requests parity record *r* and column of **P** stored at the bucket. If $l \geq k'$, then the failure is again catastrophic. *Idem*, if less than *l* parity buckets respond to the probe. Otherwise, bucket *p* produces the columns of **H** from **I** according to the offsets of the $k' - l$ data records found, completes with the columns of **P** received and performs the RS-decoding.

Bucket *p* sends to the client the recovered record *c* or the information that key *c* is not in the file. It alerts the coordinator otherwise.

4.2 Bucket Recovery

Both data and parity buckets can be recovered using RS-decoding, as long as there are *m* buckets in the segment. Parity records can be also recovered from the data buckets through RS-encoding. Record recovery can be performed concurrently with bucket recovery, as the latter is a more involved operation.

Unavailable buckets are recovered at spare servers. To start, the coordinator probes the segment with the bucket to recover for the availability of the other buckets. If *m* or more reply, then recovery proceeds, otherwise the case is catastrophic. The coordinator passes their addresses and further control of the operation to one of the spare servers. If there are data records to

recover, the spare collects columns of parity matrix **P** at the parity buckets, and forms and inverts matrix **H**. It then calculate the missing data records consecutively, stores them or sends to the spares where they should be. The key fields are from the parity records. The non-key fields are calculated from $\mathbf{H}^{-1}$, from the non-key fields of other data records in the group, and from the B-fields of the parity records.

Once data records are recovered, the remaining parity buckets to recover, if any, are produced in turn, using matrix **G**. If only parity buckets should be recovered, then one skips the reconstruction of data buckets. In a final step, the spare notifies perhaps the originator of the query of new locations through IAMs.

4.3 Key search

Normally, the LH*$_{RS}$ key search for key *c* is the LH* key search. The degraded mode is triggered by the client or the forwarding server. The coordinator uses the LH* file state parameters to calculate the address of the correct bucket. If this address is not the one of the unavailable or displaced bucket, it forwards *c* to the correct bucket. If that bucket is available, it replies to the LH*$_{RS}$ client, including the IAM. If the coordinator finds the correct bucket unavailable, it attempts the record recovery. Likewise, it initiates the recovery of any unavailable forwarding bucket.

4.4 Scan

In the normal mode the scan proceeds as a LH* scan. We recall, that the client starts with a series of unicast messages or a multicast message. The request specifies whether the scan has deterministic or probabilistic termination. In both cases, buckets that have relevant records send them to the client. For the probabilistic termination, then only these buckets reply. Otherwise, every bucket in the file replies at least with the bucket number. The LH* deterministic termination protocol detects whether every bucket has replied, even if the client image was outdated.

The degraded mode occurs only if unicast messages are used to deliver the request or if the deterministic termination is requested. The coordinator attempts the corresponding bucket recovery and the successful termination of the scan.

4.5 Insert

In normal mode, a LH*$_{RS}$ client performs the insert like a LH* client. The correct data bucket resends the record as **D**-record to all the parity buckets of the group. Their addresses are in its header.

The degraded mode is triggered by the client or the forwarding server or the correct data bucket that finds an unavailable or displaced bucket. The finder sends the record to the coordinator. For the client, the operation is then successfully terminated. The coordinator determines the correct bucket for the insertion. If it is unavailable, then it attempts the bucket recovery with the record to be inserted. As for the key search, it also recovers any unavailable forwarding bucket.

4.6 Split

As in the generic LH*, an insert to an overflowing bucket is reported to the coordinator. This usually triggers a split of the bucket pointed out by the *split pointer n* which is one of the file state parameters. Bucket *n* is typically different from that receiving the insert. After the split, $n := n + 1 \bmod 2^j$ and, if $n := 0$, then $j = j+1$. Initially, $n = j = 0$.

In normal mode, the coordinator assigns new ranks to all the records, whether they remain in the parent bucket or end up in the new bucket. The move of a record either within the parent bucket (unless by chance the new and the old rank coincide) or to new bucket is formally a deletion followed by an insert. Existing parity buckets are updated accordingly. In practice, it should be more efficient to recreate them.

If a new bucket starts a group, then the coordinator creates new parity buckets and their records. If the number of parity buckets per segment is being upgraded, the split operation appends in addition a new bucket to the group of the parent bucket.

The degraded mode consists basically of the recovery of the bucket, combined in the implementation dependent way with its split.

4.7 Update

In the normal mode, the client performs the update as for LH*. At the correct bucket, the update does not change the rank *r* of the record, unless a split occurs between the time the client reads the record and the time of the return of the update. In the former case, the bucket calculates the **D**-record and sends it with its rank *r* to the parity buckets. These buckets recalculate their parity records *r*. In the latter case, the parent bucket gets the updated record anyhow and computes whether it should migrate or stay. In both case the rank *r* typically changes. The update by definition concerns only non-key data otherwise it is a record deletions and the insert to, usually, different location. If the updated record moves, there is no more parity records concerning it in its parent segment. It suffices thus that whether the record moves or not, only the bucket that finally stores it computes the **D**-record and sends it out.

The degraded mode may start during the search for the record to update or when the client sends back the update or when the servers sends out **D**-record. If the correct bucket is available, the coordinator resends the record there. Otherwise, the coordinator updates the record in the recovered bucket. If the forwarding bucket was unavailable, the coordinator initiates the recovery of this bucket. The originator of the degraded mode gets the address of the recovered or displaced bucket in the IAM.

4.8 Deletion

In the normal mode, the client performs the deletion of record *c* as for LH*. The correct bucket in addition sends its address and record *c* with its rank *r*, to the parity buckets. Each bucket removes key *c* from its parity record *r*. If *c* is the last key, record *r* is deleted. Otherwise, its parity field *B* is adjusted.

In the degraded mode, if the data bucket is unavailable or displaced, the coordinator localizes the correct data bucket. It recovers it without the record to be deleted, as well as, perhaps, the unavailable forwarding or parity buckets.

4.9 Merge

Deletions may trigger a *bucket merge* that is the inverse to a split. In the normal mode, it moves the records of the last data bucket back into its parent bucket and removes the last bucket. The moved records receive new ranks in the parent bucket. The parity buckets of both groups are updated accordingly. If the removed bucket was the only one in its group, then the parity buckets for

this group are deleted. The number of parity buckets in the segment might also decrease. In the degraded mode, first all unavailable buckets are recovered.

5 PERFORMANCE

Detailed performance analysis is lengthy [LS99]. Table 2 summarizes basic values, intended as guidelines for the file design. The actual costs may be larger, or noticeably smaller.

5.1 Access

As usual, we measure access performance with the number of messages, as the metric independent of network speed and topology. A message contains at most one record. Table 2 shows the typical and the worst costs of various operations. For convenience, we explicitly show also the typical overhead of high-availability. The worst case of an operation accessing parity records is computed for a $(k+1)$ available group in a k-available file. The worst cost beyond the practical sense is omitted (n/a). For instance, very unlucky hashing could skyrocket the split or merge cost. The typical file has bucket capacity $b >> 1$, size $M >> m$, i.e. has several groups, and the load factor of data buckets of 0.7. It uses unicast messages, except for starting the parallel scans.

Mode	Normal			Degraded
	typical	max	overhd	
Succ. key search	2	4	0	$1+R$
Unsucc key search	2	4	0	4
Scan (det term)	$1+M$	n/a	0	$1+M+yB$
Insert	$1+k$	3+k+1	k	$1+R$
Update	$2+k$	6+k+1	k	$1+R+B$
Delete	$1+k$	3+k+1	k	$1+k+B$
Split	$0.35b$ + $+0.7bk$	n/a	$0.7\ bk$	$0.35b+0.7bk+$ $+B$
Merge	$b\ b+2b$ $+\ bk$	n/a	$2b\ bk$	$b\ b+\ 2b\ bk+$ $+B$

Mode	Typical	Max
Exist. record recovery (R)	$1+2m$	$2 + 2m + k$
Bucket recovery (B)	$0.7b\ (m+x-1)$	n/a
Storage Overhead	k / m	$(k+1) / m$

Table 2 LH*$_{RS}$ data access and high-availability performance.

The performance of key search in normal mode is that of LH*. It is independent of M and does not carry any high-availability overhead. The degraded mode cost includes the record recovery cost R. The successful search cost typically depends on m, but, perhaps surprisingly, not the unsuccessful one. The degraded mode increases the successful search time typically $(m+1)$ times. Notice that this performance is also independent of M and about best possible.

The scan also performs as for LH* and does not carry the high-availability overhead. The degraded mode carries the bucket recovery cost B, times the number y of unavailable buckets encountered in different groups.

An insert, update or delete in normal mode carries the overhead of typically k messages to parity buckets. This is the theoretical minimum for any k-available schema. The actual overhead may be also $k + 1$ when the availability starts to scale. This is also the minimal price for the scalable availability. The costs of degraded mode includes B at least.

The split and merge costs in Table 2 are easy to derive. Factor b denotes the data buckets load factor low enough to trigger merges.

Record recovery cost R in Table 2 results directly from the algorithm. The recovery starts with 1 message to the coordinator. Then, there is typically one message to a parity bucket of the failed bucket. If the searched key is not found at this bucket, there is only 1 more message to the client. Otherwise the bucket sends out typically m-1 messages to data buckets. Finally the recovered record is sent to the client.

The worst case corresponds to $k+1$ unavailable buckets probed in vain. Notice from the algorithm that there are also other cases. A group may be incomplete, with $l < m$ actual data records, hence the degraded successful search cost can be substantially under the typical one, reaching even only 4 messages. More precise estimates of R taking to the account the likelihood of each case, and of x-bucket unavailability remains to be done.

The bucket recovery cost B estimates in Table 2 result directly from the algorithm. It considers the typical bucket load of $0.7b$ records, and presence of $x \geq 1$ failures. Notice the efficiency for $x > 1$ due to the simultaneous recovery of <u>all</u> the unavailable buckets. The worst case analysis obviously does not apply here, as the worst bucket load could be assumed arbitrarily high.

Finer estimates remain to be determined. The presence of incomplete groups, likely for groups towards tops of the buckets, decreases the cost. In contrast, a parity bucket to recover should often have more records than a typical data bucket, hence a higher recovery cost. It has indeed as many records as the most loaded data bucket in the group. The deviation should obviously increase with m. It becomes more likely that some data bucket in the group has more records than the average load. The LH* file with 70% load is however known to have only a few overflow records. Hence, regardless of m, a parity bucket with more than b records in such file is unlikely as well.

Notice finally that Table 2 proves globally an excellent scalability of the LH*$_{RS}$ file manipulations. The costs are either independent of the file size M, or typically increase about as little as possible, basically through the necessary increase to k. We recall from Section 2.2 that to scale k with M is mandatory for the reliability. Section 5.4 analyzes this issue more in depth.

5.2 Storage

Each bucket group carries typically k or sometimes $k+1$ parity buckets and bucket is the storage allocation unit for both data and parity records. The file storage overhead, is thus typically k/m. This is the minimal overhead for any k-available file, regardless of the parity calculus method used [H&a94]. The additional overhead of up to $1/m$ constitutes the price for the scalable availability. This is also the minimal cost of this capability.

The overhead storage at each parity bucket server for RS calculus specific data is in practice negligible. One needs stable storage basically only for the m-element single column of **P**, and for the 2^f or $3*2^f$ elements of the log multiplication table. The inversion of the matrix **H** requires only $2m^2$ elements of temporary storage.

5.3 Parity Calculus Time

Parity encoding and decoding speed depend on the network and CPU performance. The decoding also strongly depend on the choice of the group size m whose larger values benefit the storage

overhead but make the recovery costs higher in turn. Easy but lengthy evaluations that we skip here show that on a rather typical site with 400 MHZ CPU, and 100 Mb/s network the resulting times should remain acceptable. For quite large m = 32, the record recovery time of 1KB records stored in RAM buckets is in the order of milliseconds. Assuming for instance the data bucket capacity of b = 3000 records, the bucket recovery should take less than a minute. For similar disk buckets, the time for record recovery is about a second and bucket recovery takes a few minutes.

5.4 Reliability

The *reliability* is the probability that all the data are available, i.e., that there is no catastrophic failure. It depends on b, m, k, M and the probability p that a single bucket is unavailable. One can estimate the reliability of an LH*$_{RS}$ file through formulae using these parameters developed for LH*$_{SA}$, [LMR98]. Both schemes use record grouping. Their differences influence the storage overhead and other performance factors but not the basic calculation of reliability. Same parameters lead to the same estimate.

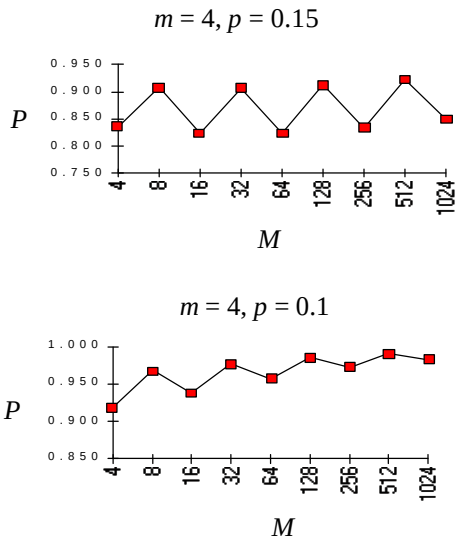

Figure 3 Uncontrolled reliability of an LH*$_{RS}$ file

Figure 3 shows two simulated curves of the reliability P for an LH*$_{RS}$ file with uncontrolled reliability obtained in this way. The values of p chosen seem conservatively realistic [S99]. They mean that a site is unavailable on the average for 3-5 days per month. Each minimum of P is at the size m^i that starts next scaling up of k for the (k+1)-availability. Each maximum is at the size $2m^i$ when the (entire) file becomes (k+1)-available and $k := k+1$. The files scale to M=1024 buckets, i.e., somewhere between 1÷100 Tbyte. In both cases, without the scalable availability, P would start continuously decreasing from M = 8, instead of remaining above the value close to the reliability of a single bucket, respectively, above 92% and 82%. For p = 0.15 the successive minima of P have the tendency to remain about the same, while for p = 0.1 they increase progressively. This tendency would be even stronger for lower p. One may then delay the increase of the availability level k with respect to the basic schema, through the reliability control. The threshold P_{min} on P value could be P_{min} = 0.9, i.e., the reliability 1-p of a single bucket. Alternatively, one may choose larger group size m.

The curve for p = 0.15 shows in contrast that the reliability evolution is close to optimal. The minimal value of P stays automatically about 0.85. For even higher p, and same m, the uncontrolled reliability would not suffice and the curve would decrease.

Both curves show that, the reliability control is useful for a multicomputer with sites characterized by $p \leq 0.1$. For less reliable sites with $p > 0.15$, it appears necessary in practice, as $m < 4$ seems the smallest useful choice. Detailed analysis in [LMR98] confirms this behavior. While not only higher p but also a larger m may make the reliability control necessary, these results show nevertheless that for $p << 0.1$, the uncontrolled reliability may suffice for a quite large m. For instance for p = 0.01, one may choose m = 16 and for p = 0.005, even m = 32 suffices to keep $P \geq 99\%$ up to M = 32K.

6 VARIANTS

An application may benefit from selected performance tuning. Specific variants of the basic schema may be designed to address this concern. First, there are numerous variants of LH* schema known, their choice impacts the performance of the LH*$_{RS}$ data file. They differ by the internal structure of buckets, the split algorithm, the strategy for the load factor control… Some variants do not even have the coordinator.

There are also issues specific to the parity management. The implementation choices for GF multiplication, including the data structures for the tables, impact the calculus speed. For instance, we can avoid the calculus modulo 2^f-1 as in Section 2.3.1, and thus increase the speed of the method. These additions or subtractions find an entry in the anti-logarithm table. If one replicates the anti-logarithm table once above and once below the table used in contiguous memory location, then the non-modulo operations suffice. The trade-off is thus to double the table size. The resulting tables have $4*2^f$ entries. In particular, $GF(2^8)$ requires then 1 KB instead of only 512 B.

Furthermore, other algorithms are known for matrix inversion and to generate matrix **G**. The internal structure of a parity bucket obviously influences its access and storage performance. The bucket recovery calculus can be made parallel between the participated buckets. The algorithms recovering from specific catastrophic failures can be added. Finally, the Reed Solomon Codes used are not the only possibility. Some other codes are potentially attractive as well, [ABC97], [H&a94], [BFT98].

The storage of parity buckets allows for interesting optimizations. The basic scheme stores parity buckets at a dedicated servers. This overhead to the number of servers may itself bother an application. Next, while the searches in normal mode do not concern the parity servers, a parity bucket is involved in every update of a data bucket in the group. The processing load from data modification at a parity server is hence about m times larger than at a data server. An application with a large amount of data modifications could see the parity servers becoming bottlenecks.

The correctness of the parity calculus does not depend on parity buckets being stored at separate servers. It merely requires that no server contains two buckets in the same segment. To better balance the load, one may replace then a single parity bucket with m buckets storing b/m records each, stored the different pieces at different servers. The m buckets can form a simple hash subfile.

This variant equalizes the load from of the data modifications. On the negative side, it requires more parity servers and sees more messaging during splits and reconstructions.

To decrease the number of parity servers, one may share a server between a data and a parity bucket. One simple rule locates the i^{th} parity bucket of group j with the i^{th} data bucket of group $j+1$. Figure 2 may be seen as illustrating this rule. It guarantees that all buckets in a segment are stored at different server. It can be easily extended to the above discussed parity subfiles. In this scheme, every server carries at most one data bucket and zero, or some parity buckets. The servers carrying data for the first group will never have parity data. In turn, some servers to serve the data buckets of the next group to be generated carry only parity buckets, but not yet data buckets. The "dark size of the Moon" is obviously increased storage use at each server and processing load.

Finally, one can have the servers for data of group 0 temporarily carrying the parity buckets for the last existing group. If the file expands further, these buckets move to the adequate locations, being replaced by new last buckets. This simple loop-back strategy eliminates the additional parity servers entirely. It minimizes the number of servers for the file to M, required anyway for the data buckets. Notice the potential interest of this variant to the users of the current parallel DBMSs. Their currently 0-available hash or even range-partitioning methods, could be enhanced to the high-availability at no additional hardware cost.

7 RELATED WORK

There were countless high-availability schemes for a single site, usually 1-available and using some RAID-like striping. A few schemes appeared for the (static) $k > 1$ k-availability in this context, [BM93], [BBM93], [H&a94], and [ABC97] recently. There were also studies for the distributed environment, e.g. [SG90] showing the inefficiency of any trivial striping. Deeper discussion of all these schemes, including SDDS schemes with mirroring or replication mentioned in the Introduction, is in [LMR98]. However, besides the LH*$_{RS}$, the only schemes known to satisfy all our goals, including the moderate storage overhead for the high-availability, are the other LH* schemes using the record grouping mentioned in the Introduction. Their mutual comparison appears as follows [LS99].

LH*$_{RS}$ may offer substantially lower overhead than LH*$_{SA}$. The reason is that the number k of parity records to make a group k-available is always exactly the theoretical minimum k/m. This is a remote consequence of the MDS property of RS-codes, [MS97].

LH*$_{RS}$ record recovery cost should typically be lower than that of LH*$_g$. It may be higher or 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, on the other hand.

Variants minimizing the number of the file servers through sharing of data and parity buckets are known only for LH*$_{RS}$. Such variant seem at best more difficult to design for the two other schemes.

If 1-availability suffices, then LH*$_g$ has the smallest split cost. Its record groups are location independent and there is no need to recalculate the parity data during splits. Generalizing the parity calculus to RS-codes allows perhaps for a k-available variant of LH*$_g$ retaining that property [LMRS99].

Finally, LH*$_{SA}$ may often recover from l-bucket failure where $l > k$ which would be catastrophic for LH*$_{RS}$. The difference may be quite substantial. For instance, a 2-available LH*$_{SA}$ file may recover records from any $l >> 2$ unavailable buckets in the same group. LH*$_{RS}$ can accommodate at most 3 unavailable buckets per group in a 2-available file, and only, provided it started to build the 3-availability. Notice that LH*$_{RS}$ also has good cases, although the overall balance seems in favor of LH*$_{SA}$. For instance, in our example file with record group size $m = 4$, made 2-available, we can recover from the unavailability of buckets 0,1,4, and 5. This failure is unrecoverable in a 2-available LH*$_{SA}$ file with the same groups.

8 CONCLUSION

LH*$_{RS}$ schema uses the concept of record grouping and the Reed Salomon codes to provide scalable, distributed and high-availability files, badly needed by modern applications. Its interesting properties, including the scalable availability, near-optimal access performance and storage use efficiency, should prove attractive. The schema offers distinct advantages over the other high-availability schemes known.

Among potential applications, there are modern database systems, that need continuously larger scalable databases, and for which the parallel access is already a must [FBW97], [B&al95], [IBM99]. Many of the existing databases or warehouses grow very rapidly. The well-known UPS multidatabase 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. This is the case of the 166–site multicomputer of Inktomi at Santa Clara, CA, and of the 100-site of Yahoo in Vienna, VA, which is also built by Inktomi, [I98]. An implementation of SDDSs is under study for such applications [G99]. For all these needs, both scalability and 24/7 availability are critical. The already mentioned mishap of E-Bay is here to stay as the reminder.

Future work should include the prototype implementation and deeper analysis of various design issues, as well as of performance factors discussed in the related sections. This applies also to the variants. These goals start to be addressed for Wintel multicomputers, [L00]. The prototype described confirms the feasibility of the schema for that environment, and seems to perform as expected in Section 5.

On the other hand, one should port the RS parity schema to other known 0-availaible SDDS schemes. This should especially concern the RP*schema. Finally, one should study the other erasure correcting codes referred to in Section 6.

9 ACKNOWLEDGEMENTS

This research was partly sponsored by a Grant of IBM Almaden Res. Cntr., Storage Systems Div., and by a Grant of Microsoft Research. We also thank Mario Blum, Walter Burkhard, Jim Gray, Mattias Ljungström, Jai Menon, and Tore Risch for helpful discussions.

APPENDIX A

Let n be the maximal segment size and m the maximal group size. We recall that the generator matrix **G** of an RS-code has m rows and n columns. The left m x m submatrix of **G** is the identity matrix **I**, since we use a *systematic* RS code. Any square

submatrix formed from any m different columns of **G** is invertible.

We derive matrix **G** from a Vandermonde matrix **V**. **V** has m rows and n columns, these we index by the n elements $g_j \in GF(n)$, $j = 0, ..., n\text{-}1$. We number the columns and rows from 0 and order the field elements so that $g_0 = 0$ and $g_1 = 1$. Coefficient $v_{i,j}$ located in the i^{th} row and the j^{th} column of **V** is then defined to be the j^{th} element of the Galois Field raised to the i^{th} power that is:

$$v_{i,j} = g_j^i.$$

Thus,

$$\mathbf{V} = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots \\ g_0 & g_1 & g_2 & g_3 & \\ g_0^2 & g_1^2 & g_2^2 & g_3^2 & \\ g_0^3 & g_1^3 & g_2^3 & g_3^3 & \\ \vdots & & & & \ddots \end{pmatrix}$$

Row 0 contains only ones since for every j, $g_j^0 = 1$. Since $g_0 = 0$, the first column contains otherwise zeroes. Since $g_1 = 1$, the 2nd column contains only 1's. Vandermonde showed the determinant of any square submatrix of **V** consisting of m columns generated by elements g_i to be

$$\det \mathbf{V} = \prod_{i<j} g_j - g_i \neq 0.$$

It follows that any square m by m submatrix of **V** is invertible. This property holds also for the *extended* V with last column 0, 0, ... 0, 1 that we used to generate our example **G** in Section 2.3.4. We now give the details of our transformation of **V** into **G = I|P** where **I** is the identity matrix. We denote **m**[i,j], $i,j \geq 0$, the coefficient of matrix **m** in row i and column j. We denote the j^{th} row of the current matrix with $\mathbf{m}_j$. We use *elementary row transformation* [MM92]. These are multiplying a row by a scalar, exchanging two rows, and adding a multiple of one row to another. We denote these transformations by $\mathbf{m}_j \Leftarrow a\mathbf{m}_j$, $\mathbf{m}_j \Leftrightarrow \mathbf{m}_i$, $\mathbf{m}_j \Leftarrow \mathbf{m}_j + a\mathbf{m}_i$, $a \in$ GF(2^f) respectively. Our algorithm uses up to m row transformations to transform a column into a unit vector. The first column is already the first unit vector. The second column has already the one in position [1,1], and we add the second row to all the other rows resulting in the second unit vector for the second column. This operation retains the form of the first column. We now change the third column into the unit vector. The diagonal element **m**[2,2] there is obviously non-zero. We multiply the third row with the inverse of this element, so that the coefficient **m**[2,2] is now 1. Then we generate zeroes in the third column by adding **m**[i,2]$\mathbf{m}_2$ to all other rows $\mathbf{m}_i$. This operation does not change the first and second column. Continuing in this manner, we transform m left columns of **V** into unit vectors, i.e. the **I** submatrix. We give pseudo-code, à la [PTVF92], in Figure 4:

```
    Initialize m = V;
for all columns i = 0, ..., m-1 do
        {
(4)     mi ⇐ m[i,i]-1mi;
        for all rows j = 0, ..., m-1, but j≠i, do
                mj ⇐ mj – m[j,i] mi;
        }
```

Figure 4 Pseudo-code to transform V **into generator matrix G**.

Our inversion algorithm proceeds similarly. We form the m x $2m$ matrix **H|I** from the m x m invertible matrix **H**. We transform **H|I** into $\mathbf{I}|\mathbf{H}^{-1}$ using the algorithm in Figure 4, with one exception. As the diagonal element **m**[i,i] in line (4) may be zero, we replace line (4) with :

```
(4a)    if m[i,i] = = 0 do
(4b)    {
(4c)      find a j > i such that m[j,i] ≠ 0;
(4d)      mj ⇔ mi;
(4e)    }
(4f)    mi ⇐ m[i,i]-1mi;
```

REFERENCES

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

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

[B99] Bartalos, G. Internet: D-Day at eBay. Yahoo INDIVIDUAL INVESTOR ONLINE, (Jul 19, 1999).

[B99a] Bertino & al. Indexing Techn. for Advanced Database Systems. Kluver, 1999.

[BBM93] Blaum, M., Bruck, J., Menon, J. EVENODD: An Efficient Scheme for Tolerating Double Disk Failures in RAID Architectures. IEEE Trans. on Computers, Vol. C-44, No. 2, pp. 192-202, February 1995.

[BFT98] Blaum, M & al. Array Codes, Handbook of Coding Theory. V.S. Pless and W.C. Huffman, (ed.), Elsevier Science B.V., 1998.

[BV98] Breitbart, Y. Vingralek, R. Distributed and Scalable B+ tree Data Structures. Workshop on Distr. Data and Struct., 1998, Carleton Scientific (publ.)

[BM93] W. A. Burkhard, J. Menon: Disk Array Storage System Reliability. 22rd Intl. Symp. on Fault Tolerant Computing, Toulouse, June 1993, 432-441.

[CACM97] Special Issue on High-Performance Computing. Comm. Of ACM. (Oct. 1997).

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

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

[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.

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

[IBM99] Breaking the Scalability Barrier on Windows NT. http://www.software.ibm.com/data/pubs/papers/nt-scale/

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

[KLR96] J. Karlson, W. Litwin, T. Risch. LH*LH: A Scalable High Performance Data Structure for Switched Multicomputers. Extending Database Technology, EDBT96, Springer Verlag.

[L80] W. Litwin. Linear Hashing: A New Tool for File and Table Addressing. Reprint from VLDB80 in Readings in Databases, M. Stonebraker, 2nd Edition, Morgan Kaufmann Publishers, 1994.

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

[L&al97] Litwin, W., Neimat, M.-A. Levy, G., Ndiaye, S., Seck, T. LH*$_S$: a high-availability and high-security Scalable Distributed Data Structure. IEEE-Res. Issues in Data Eng. (RIDE-97), 1997.

[L00] Ljungström, M. Implementing LH*$_{RS}$: A Scalable Distributed High-Availability Data Structure. Master Th. (Feb. 2000), CS Dep., U. Linkoping, Suede.

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

[LMRS99] Litwin, W., Menon, J.Risch, T., Schwarz, Th. Design Issues For Scalable Availability LH* Schemes with Record Grouping. DIMACS Workshop on Distributed Data and Structures. Carleton Scientific, 1999.

[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 Trans. on Database Systems, Dec. 1996.

[LN96] W. Litwin, M.-A. Neimat: "High-Availability LH* Schemes with Mirroring", Intl. Conf. on Coop. Inf. Systems, (COOPIS). IEEE Press 1996.

[LR97] Litwin W., Risch, T. LH*g: a High-Availability Scalable Distributed Data Structure through Record Grouping. Res. Rep. CERIA, U. Dauphine & U. Linkoping (May. 1997).

[LS99] Litwin, W., Schwarz, Th. LH*$_{RS}$: A High-Availability Scalable Distributed Data Structure using Reed Solomon Codes. Res. Rep. CERIA, U. Dauphine (Sept. 1999).

[M97] Gates, B. The Microsoft Scalability Day http://204.203.124.10/backoffice/scalability/coverage.htm

[MM92] Marcus, M., Minc, H. A Survey of Matrix Theory and Matrix Inequalities, Dover, New York, 1992.

[P98] President's Inf. Techn. Advisory Comm. Interim Rep. To the Pres. Of the United States. August 1998.

[MS97] MacWilliams, F. J., Sloane, N. J. A. The Theory of Error Correcting Codes, Elsevier / North Holland, Amsterdam, 1997.

[PTVF92] Press, W. H., Teukolsky, S. A., Vetterling W. T., Flannery, B. P. Numerical Recipes in C: The Art of Scientific Computation, 2nd ed., Cambridge University Press, 1992.

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

[S99] Smith, D. The Cost of Lost Data. Res. Rep. School of Business and Management, Pepperdine University, 1999.

[SDDS] SDDS-bibliography. http://ceria.dauphine.fr/SDDS-bibliograhie.html

[SG90] M. Stonebraker, G. Schloss: "Distributed RAID – A New Multiple Copy Algorithm", 6th Intl. IEEE Conf. on Data Engineering, 1990, IEEE Press, pp. 430-437.

[VBW94] Vingralek R., Breitbart Y., G. Weikum. Distributed File Organization with Scalable Cost/Performance. ACM-SIGMOD Intl. Conf. on Management of Data, 1994, 253-264.

[VBW98] Vingralek R., Breitbart Y., Weikum G. SNOWBALL: Scalable Storage on Networks of Workstations with Balanced Load. Distr. and Par. Databases, 6, 2, 1998.