Online Appendix to: LH*RS—A Highly-Available Scalable Distributed Data Structure

WITOLD LITWIN and RIM MOUSSA
Université Paris Dauphine
and
THOMAS SCHWARZ, S. J.
Santa Clara University

APPENDIX C

8. LH* ADDRESSING ALGORITHMS

The LH*RS *Forwarding Algorithm* executed at bucket a, getting a key-based request from a client is as follows [Litwin et al. 1996]. Here c is the key and j is the level of bucket a. As the result, the bucket determines whether it is the correct one for c, according to (LH), or forwards the request to bucket $a' > a$.

Algorithm A1
$a' := \mathrm{h}_j\,(c)$;
if $a' = a$ **then** accept c ;
else $a'' := h_{j-1}\,(c)$;
  if $a'' > a$ **and** $a'' < a'$ **then** $a' := a''$;
send c to bucket a';

The *Image Adjustment Algorithm* that LH*RS client executes to update its image when the IAM comes back is as follows. Here a is the address of the last bucket to forward the request to the correct one, and j is the level of bucket a. These values are in IAM. Notice that they come from a different bucket than that considered in Litwin et al. [1996]. The latter was the first bucket to receive the request. The change produces the image whose extent is closer to the actual one in many cases. The search for key $c = 60$ in the file in Figure 1(b) illustrates one such case.

Algorithm A2
if $j > i'$ **then** $i' := j - 1, n' := a + 1$;
if $n' \geq 2^{i'}$ **then** $n' = 0, i' := i' + 1$;

9. GALOIS FIELD CALCULATIONS

Our *GF* has 2^f elements, $f = 1, 2 \ldots$, called *symbols*. Whenever the *size* 2^f of a *GF* matters, we write the field as $GF(2^f)$. Each symbol in $GF(2^f)$ is a bit-string of length f. One symbol is zero, written as 0, and consists of f zero-bits. Another is the one symbol, written as 1, with $f-1$ bits 0 followed by bit 1.

Table V. Logarithms for $GF(256)$

El.	Log	El.	Log	El.	Log	El.	Log	El.	Log	El.	Log	El.	Log	El.	Log
—	—	10	4	20	5	30	29	40	6	50	54	60	30	70	202
1	0	11	100	21	138	31	181	41	191	51	208	61	66	71	94
2	1	12	224	22	101	32	194	42	139	52	148	62	182	72	155
3	25	13	14	23	47	33	125	43	98	53	206	63	163	73	159
4	2	14	52	24	225	34	106	44	102	54	143	64	195	74	10
5	50	15	141	25	36	35	39	45	221	55	150	65	72	75	21
6	6	16	239	26	15	36	249	46	48	56	219	66	126	76	121
7	198	17	129	27	33	37	185	47	253	57	189	67	110	77	43
8	3	18	28	28	53	38	201	48	226	58	241	68	107	78	78
9	223	19	193	29	147	39	154	49	152	59	210	69	58	79	212
A	51	1a	105	2a	142	3a	9	4a	37	5a	19	6a	40	7a	229
B	238	1b	248	2b	218	3b	120	4b	179	5b	92	6b	84	7b	172
C	27	1c	200	2c	240	3c	77	4c	16	5c	131	6c	250	7c	115
D	104	1d	8	2d	18	3d	228	4d	145	5d	56	6d	133	7d	243
E	199	1e	76	2e	130	3e	114	4e	34	5e	70	6e	186	7e	167
F	75	1f	113	2f	69	3f	166	4f	136	5f	64	6f	61	7f	87
80	7	90	227	a0	55	b0	242	c0	31	D0	08	e0	203	F0	79
81	112	91	165	a1	63	b1	86	c1	45	D1	161	e1	89	F1	174
82	192	92	153	a2	209	b2	211	c2	67	D2	59	e2	95	F2	213
83	247	93	119	a3	91	b3	171	c3	216	D3	82	e3	176	F3	233
84	140	94	38	a4	149	b4	20	c4	183	D4	41	e4	156	F4	230
85	128	95	184	a5	188	b5	42	c5	123	D5	157	e5	169	F5	231
86	99	96	180	a6	207	b6	93	c6	164	D6	85	e6	160	F6	173
87	13	97	124	a7	205	b7	158	c7	118	D7	170	e7	81	F7	232
88	103	98	17	a8	144	b8	132	c8	196	D8	251	e8	11	F8	116
89	74	99	68	a9	135	b9	60	c9	23	D9	96	e9	245	F9	214
8a	222	9a	146	aa	151	Ba	57	ca	73	da	134	ea	22	fa	244
8b	237	9b	217	ab	178	Bb	83	cb	236	db	177	eb	235	fb	234
8c	49	9c	35	ac	220	Bc	71	cc	127	dc	187	ec	122	fc	168
8d	197	9d	32	ad	252	Bd	109	cd	12	dd	204	ed	117	fd	80
8e	254	9e	137	ae	190	Be	65	ce	111	de	62	ee	44	fe	88
8f	24	9f	46	af	97	Bf	162	cf	246	df	90	ef	215	ff	175

Symbols can be added $(+)$, multiplied $(\cdot)$, subtracted $(-)$ and divided $(/)$. These operations in a *GF* possess the usual properties of their analogues in the field of real or complex numbers, including the properties of 0 and 1. As usual, we may omit the multiplication symbol.

Initially, we elaborated on the LH^*_{RS} scheme for $f = 4$ [Litwin and Schwarz 2000]. First experiments showed that $f = 8$ was more efficient. The reason was the (8-bit) byte and word-oriented structure of current computers [Liungström 2000]. Later, the choice of $f = 16$ turned out to be even more practical and became our final choice, see Section 5. For didactic purposes, we discuss our parity calculus nevertheless for $f = 8$, that is, for $GF(2^8) = GF(256)$. The reason is the sizes of the tables and matrices involved. We call this *GF* F. The symbols of F are all the byte values. F thus has 256 symbols which are $0, 1 \ldots 255$ in decimal notation, or $0, 1 \ldots$ ff in hexadecimal notation. We use the latter in Table V and often in our examples.

The addition and the subtraction in any of our $GF(2^f)$ are the same. These are the bit-wise XOR (Exclusive-OR) operation on f-bit bytes or words. That is

$$a + b = a - b = b - a = a \oplus b = a \text{ XOR } b.$$

The XOR operation is widely available, for example, as the $\wedge$ operator in C and Java, that is, a XOR $b = a \wedge b$. The multiplication and division are more involved. There are different methods for their calculus. We use a variant of the *log/antilog* table calculus [Litwin and Schwarz 2000; McWilliams and Sloane 1997].

The calculus exploits the existence of primitive elements in every GF. If α is primitive, then any element $\xi \neq 0$ is α^i for some integer power i, $0 \leq i < 2^f - 1$. We call i the *logarithm* of ξ and write $i = \log_\alpha(\xi)$. Table V tabulates the nonzero $GF(2^8)$ elements and their logarithms for $\alpha = 2$. Likewise, $\xi = \alpha^i$ is then the *antilogarithm* of i, and we write $\xi = \text{antilog}(i)$.

The successive powers α^i for any i, including $i \geq 2^f - 1$, form a cyclic group of order $2^f - 1$ with $\alpha^i = \alpha^{i'}$ exactly if $i' = i \bmod 2^f - 1$. Using the logarithms and the antilogarithms, we can calculate multiplication and division through the following formulas. They apply to symbols $\xi, \psi \neq 0$. If one of the symbols is 0, then the product is obviously 0. The addition and subtraction in the formulas is the usual one of integers:

$$\begin{aligned} \xi \cdot \psi &= \text{antilog}(\log(\xi) + \log(\psi) \bmod (2^f - 1)), \\ \xi / \psi &= \text{antilog}(\log(\xi) - \log(\psi) + 2^f - 1 \bmod (2^f - 1)). \end{aligned}$$

To implement these formulas, we store symbols as char type (byte long) for $GF(2^8)$ and as short integers (2-bytes long) for $GF(2^{16})$. This way, we use them as offsets into arrays. We store the logarithms and antilogarithms in two arrays. The logarithm array log has 2^f entries. Its offsets are symbols 0x00 . . . 0xff, and entry i contains $\log(i)$, an unsigned integer. Since element 0 has no logarithm, that entry is a dummy value such as 0xffffffff. Table V shows the logarithms for F.

Our multiplication algorithm applies the antilogarithm to sums of logarithms modulo $2^f - 1$. To avoid the modulus calculation, we use all possible sums of logarithms as offsets. The resulting antilog array then stores antilog[i] = antilog($i \bmod (2^f - 1)$) for entries i = 0, 1, 2 . . . , $2(2^f - 2)$. We double the size of the antilog array in this way to avoid the modulus calculus for the multiplication. This speeds up both encoding and decoding times. We could similarly avoid the modulo operation for the division as well. In our scheme, however, division is rare and the savings seem too minute to justify the additional storage (128KB for our final choice of $f = 16$). Figure 14 gives the pseudo-code generating our `log` and `antilog` multiplication table (Table V). Figure 13 shows our final multiplication algorithm. We call them respectively `log` and `antilog` arrays. The following example illustrates their use.

```
GFElement mult (GFElement left,GFElement right) {
    if(left==0 || right==0) return 0;
    return antilog[log[left]+log[right]];
}
```

Fig. 13. Galois Field multiplication algorithm.

```
#define EXPONENT           16  // 16 or 8
#define NRELEMS           (1 << EXPONENT)
#if ((EXPONENT == 16)
   #define CARRYMASK           0x10000
   #define POLYMASK            0x1100b
#elif (EXPONENT == 8)
   #define CARRYMASK          0x100
   #define POLYMASK            0x11d
#endif
void generateGF()
{
   int i;
   antigflog[0] = 1;
   for (i = 1; i < NRELEMS; i++)  {
      antigflog[i] = antigflog[i-1] << 1;
      if(antigflog[i] & CARRYMASK) antigflog[i] ^= POLYMASK;}
   gflog[0] = -1;
   for(i = 0;i < NRELEMS-1;i++)
      gflog[antigflog[i]] = i;
   for(i = 0;i < NRELEMS-1;i++)
      antigflog[NRELEMS-1+i]= antigflog[i];
}
```

Fig. 14. Calculus of tables `log` and `antilog` for $GF(2^f)$.

Example 7.

$$
\begin{aligned}
&45 \cdot 1 + 49 \cdot 1a + 41 \cdot 3b + 41 \cdot ff \\
&= 45 + \text{antilog}(\log(49) + \log(1a)) + \text{antilog}(\log(41) + \log(3b)) \\
&\quad + \text{antilog}(\log(41) + \log(ff)) \\
&= 45 + \text{antilog}(152 + 105) + \text{antilog}(191 + 120) + \text{antilog}(191 + 175) \\
&= 45 + \text{antilog}(257) + \text{antilog}(311) + \text{antilog}(191 + 175) \\
&= 45 + \text{antilog}(2) + \text{antilog}(56) + \text{antilog}(111) \\
&= 45 + 04 + 5d + ce \\
&= d2
\end{aligned}
$$

The first equality uses our multiplication formula but for the first term. We use the logarithm array `log` to look up the logarithms. For the second term, the logarithms of 49 and 1a are 152 and 105 (in decimal), respectively (Table V). We add these up as integers to obtain 257. This value is not in Table V, but `antilog[257]= 4`, since logarithms repeat the cycle of mod $(2^f - 1)$ that yields here 255. The last equation sums up four addends in the Galois field which in binary are 0100 0101, 0000 0100, 0101 1101, and 1100 1110. Their sum is the XOR of these bit strings, yielding 1101 0010 = d2.

To illustrate the division, we calculate 1a / 49 in the same *GF*. The logarithm of 1a is 105, the logarithm of 49 is 152. The integer difference is –47. We add 255, obtain 208, hence read `antilog[208]`. According to Table V it contains 51 (in hex) which is the final result.

10. ERASURE CORRECTION

LEMMA FOR SECTION 3.2. *Let* m *be the current group size, and* m′ *be the generic group size. Denote the generic parity matrix with* $\mathbf{G}'$. *Form a matrix* $\mathbf{G}$, *as in Section* 3.2, *from the first* m *rows of* $\mathbf{G}'$, *but leaving out the resulting zero columns* $\mathrm{m}+1, \ldots \mathrm{m}'$. *Then we obtain the same encoding and decoding whether we use* $\mathbf{G}$ *and the first* m *data buckets, or whether we use* $\mathbf{G}'$ *and* m *data buckets but with all but the first* m′ *data buckets virtual zero buckets.*

PROOF. Consider that for the current group size $m < m'$. There are $m' - m$ dummy data records, padding each record group to size m'. Let $\mathbf{a}$ be the vector of m' symbols with the same offset in the data records of the group. The rightmost $m' - m$ coefficients of $\mathbf{a}$ are all zero. We can write $\mathbf{a} = (\mathbf{b} \mid \mathbf{o})$, where $\mathbf{b}$ is an m-dimensional vector and $\mathbf{o}$ is the $m' - m$ dimensional zero vector. We split $\mathbf{G}'$ similarly by writing

$$\mathbf{G}' = \begin{pmatrix} \mathbf{G}_0 \\ \mathbf{G}_1 \end{pmatrix}.$$

Here $\mathbf{G}_0$ is a matrix with m rows, and $\mathbf{G}_1$ is a matrix with $m' - m$ rows. We have $\mathbf{u} = \mathbf{a} \cdot \mathbf{G} = \mathbf{b} \cdot \mathbf{G}_0 + \mathbf{o} \cdot \mathbf{G}_1 = \mathbf{b} \cdot \mathbf{G}_0$. Thus, we only use the first m coefficients of each row for encoding.

Assume now that some data records are unavailable in a record group, but m records among $m + k$ data and parity records in the group remain available. We can now decode all the m data records of the group as follows. We assemble the symbols with offset l from the m available records in a vector $\mathbf{b}^l$. The order of the coordinates of $\mathbf{b}^l$ is the order of columns in $\mathbf{G}$. Similarly, let $\mathbf{x}^l$ denote the word consisting of m data symbols with the same offset l from m data records in the same order. Some of the values in $\mathbf{x}^l$ are from the unavailable buckets and thus unknown. Our goal is to calculate $\mathbf{x}$ from $\mathbf{b}$.

To achieve this, we form an m' by m' matrix $\mathbf{H}'$ with, at the left, the m columns of $\mathbf{G}'$ corresponding to the available data or parity records, and then the $m' - m$ unit vectors formed by the column from the $\mathbf{I}$ portion of $\mathbf{G}'$ corresponding to the dummy data buckets. This gives $\mathbf{H}'$ a specific form

$$\mathbf{H}' = \begin{pmatrix} \mathbf{H} & \mathbf{O} \\ \mathbf{Y} & \mathbf{I} \end{pmatrix}.$$

Here, $\mathbf{H}$ is an m by m matrix, $\mathbf{Y}$ an $m' - m$ by m matrix, $\mathbf{O}$ the m by $m' - m$ zero matrix, and $\mathbf{I}$ is the $m' - m$ by $m' - m$ identity matrix. Let $(\mathbf{x}^l \mid 0)$ and $(\mathbf{b}^l \mid 0)$ be the m' dimensional vector consisting of the m coordinates of $\mathbf{x}^l$ and $\mathbf{b}^l$, respectively, and $m' - m$ zero coefficients.

$$(\mathbf{x}|\mathbf{o}) \begin{pmatrix} \mathbf{H} & \mathbf{O} \\ \mathbf{Y} & \mathbf{I} \end{pmatrix} = (\mathbf{b}|\mathbf{o}).$$

That is,

$$\mathbf{xA} = \mathbf{b}.$$

According to a well-known theorem of linear algebra, for matrices of this form $\det(\mathbf{H}') = \det(\mathbf{H}) \cdot \det(\mathbf{I}) = \det(\mathbf{H})$. So $\mathbf{H}$ is invertible since $\mathbf{H}'$ is. The last equation tells us that we only need to invert the m-by-m matrix $\mathbf{H}$. This is precisely the desired submatrix $\mathbf{H}$ cut out from the generic one. This concludes our proof. □

11. ADDITIONAL LH^*_{RS} FILE MANIPULATIONS

We now describe the operations on the LH^*_{RS} file that were yet not presented, as less relevant to the high-availability or less frequent. These are file creation and removal, key search, nonkey search (scan), record delete, and bucket merge.

11.1 File Creation and Removal

The client creates an LH^*_{RS} file F as an empty data bucket 0. File creation sets the parameters m and K. The latter is typically set to $K = 1$. The SDDS manager at bucket 0 becomes the coordinator for F. The coordinator initializes the file state to $(i = 1, n = 0)$. The coordinator also creates K empty parity buckets to be used by the first m data buckets which will form the first bucket group. The coordinator stores column i of $\mathbf{P}$ with the ith parity bucket with the exception of the first parity bucket (using $\mathbf{P}'$ to generate these columns). There is no degraded mode for the file creation operation. Notice, however, that the operation fails if no $K + 1$ available servers are to be found.

If the application requests the removal of the file, the client sends the request to the coordinator. The coordinator acknowledges the operation to the client. It also forwards the removal message to all data and parity buckets. Every node acknowledges it. The unresponsive servers enter an error list to be dealt with beyond the scope of our scheme.

11.2 Key Search

In normal mode, the client of LH^*_{RS} searches for a key using the LH^* key search as presented in Section 2.1.2. The client or the forwarding server triggers the degraded mode if it encounters an unavailable bucket, called a_1. It then passes the control to the coordinator. The coordinator starts the recovery of bucket a_1. It also uses the LH^* file state parameters to calculate the address of the correct bucket for the record, call it bucket a_2. If $a_2 = a_1$, the coordinator also starts the record recovery. If $a_2 \neq a_1$, and bucket a_2 was not found to be unavailable during the probing phase of the bucket a_1 recovery, then the coordinator forwards c to bucket a_2. If bucket a_2 is available, it replies to the LH^*_{RS} client as in the normal mode, including the IAM. If the coordinator finds it unavailable and the bucket is not yet recovered, for example, is in another group than bucket a_1, then the coordinator starts the recovery of bucket a_2 as well. In addition, it performs record recovery.

11.3 Scan

A scan returns all records in the file that satisfy a certain query, Q, in their nonkey fields. A client performing a scan sends Q to all buckets in the *propagation* phase. Each server executes Q and sends back the results during the *termination* phase. The termination can be *probabilistic* or *deterministic* [Litwin et al. 1996]. The choice is up to the application.

Scan Propagation. The client sends Q to all the data buckets in its image, using unicast or broadcast when possible. Unicast messages only reach the buckets in the client image. LH*RS then applies the following LH* *scan propagation* algorithm in the normal mode. The client sends Q with the *message level* j' attached. This is the presumed level j of the recipient bucket, according to the client image. Each recipient bucket executes Algorithm A4. A4 forwards Q recursively to all the buckets that are beyond the client image. Any of these must result, perhaps recursively through its parents, also beyond the image, in a split of exactly one of the buckets in the image.

Algorithm A4: Scan Propagation

The client executes:
n' = split pointer of client.
i' = level of client.
for $a = 0, \ldots 2^{i'}+n'$ **do :**
  if $(a < 2^{i'}$ **and** $n' \leq a)$ **then** $j' = i'$ **else** $j' = i' + 1$.
  send (Q, j′) **to** a.
Each bucket a executes upon receiving (Q, j'):
j = level of a.
while $(j' < j)$ **do:**
  $j' = j + 1$;
forward (Q, j') **to bucket** $a + 2^{j'-1}$.

In normal mode, Algorithm A4 guarantees that the scan message arrives at every bucket exactly once [Litwin et al. 1996]. We detect unavailable buckets and enter degraded mode in the termination phase.

Example 8. Assume that the file consists of 12 buckets 0, 1, ... 11. The file state is $n = 4$ and $i = 3$. Assume also that the client still has the initial image $(n', i') = (0, 0)$. According to this image, only bucket 0 exists. The client sends only one message $(Q, 0)$ to bucket 0. Bucket 0 sends messages $(Q, 1)$ to bucket 1, $(Q, 2)$ to bucket 2, $(Q, 3)$ to bucket 4, and $(Q, 4)$ to bucket 8. Bucket 1 receives the message from bucket 0 and sends $(Q, 2)$ to bucket 3, $(Q, 3)$ to bucket 5, $(Q, 4)$ to bucket 9. Bucket 2 sends $(Q, 3)$ to 6 and $(Q, 4)$ to 11. Bucket 3 receives $(Q, 2)$ and forwards with level 3 to bucket 7 and with level 4 to bucket 11. The remaining buckets receive messages with a message level equal to their own level and do not forward.

Scan Termination. A bucket responds to a scan with *probabilistic termination* only if it has a relevant record. The client assumes that the scan has successfully terminated if no message arrives after a timeout, following the last reply. A scan with probabilistic termination does not have the degraded mode. The operation cannot always discover the unavailable buckets.

In *deterministic termination* mode, every data bucket sends at least its level j. The client can then calculate whether all existing buckets have responded. For this purpose, the client maintains a list L with every j received. It also maintains the count N' of replies received. The client terminates Q normally if and only if it eventually meets one of the termination conditions

(i) All levels j in L are equal, and $N' = 2^j$.

(ii) There are two levels from consecutive buckets in the list such that $j_{a-1} = j_a + 1$ and $N' = 2^{j_a} + j_a$.

Each condition determines, in fact, the actual file size N and compares N' to N. Condition (i) applies if the split pointer n is 0. Condition (ii) corresponds to $n > 0$ and, in fact, determines n as a fulfilling $j_{a-1} = j_a + 1$. The conditions on N' test that the all N buckets are answered. Otherwise, the client waits for further replies.

A scan with deterministic termination enters degraded mode when the client does not meet the termination conditions within a timeout period. The client sends the scan request and the addresses in L to the coordinator. From the addresses and the file state, the coordinator determines unavailable buckets. These may be in different groups. If no catastrophic loss has occurred in a group, the coordinator initiates all recoveries as in Section 4.1. Once they are all completed, the coordinator sends the scan to the recovered data buckets. The client waits until the scan completes in this way. Whether the termination is normal or degraded, the client finally updates its image and perhaps the location data.

Example 9. For change, we consider now a file in state $(n, i) = (0, 3)$, hence with 8 buckets $0, 1 \ldots 7$. The record group size is $m = 4$, and the intended availability level K is $K = 1$. The client image is $(n', i') = (2, 2)$. Accordingly, the bucket knows of buckets 0, 1, 2, 3, 4, and 5. The client issues a scan Q with the deterministic termination. It got replies with bucket levels j from buckets 0, 1, 2, 4, 5, and 6. None of the termination conditions are met. Condition (i) fails because, among others, $j_0 > j_4$. Likewise, condition (ii) cannot become true until bucket 3 replies. The client waits for further replies.

Assume now that no bucket replied within the timeout. The client alerts the coordinator and sends Q and the addresses in list $L = \{0, 1, 2, 4, 5, 6\}$. Based on the file state and L, the coordinator determines that buckets 3 and 7 are unavailable. Since the loss is not catastrophic (for $m = 4$ and $k = 1$ in each group concerned), the coordinator now launches recovery of buckets 3 and 7. Once this has succeeded, the coordinator sends the scan to these buckets. Each of them finally sends its reply with its j_a, perhaps some records, and its (new) address. The client adjusts its image to $(n', i') = (0, 3)$ and refreshes the location data for buckets 3 and 7.

11.4 Delete

In the normal mode, the client performs the delete of record c as for LH*. In addition, the correct bucket sends the Δ-record, the rank r of the deleted record, and key c to the k parity buckets. Each bucket confirms the reception and

removes key c. If c is the last actual key in the list, then the parity bucket deletes the entire parity record r. Otherwise, it adjusts the B-field of the parity record to reflect that there is no more record c in the record group.

The data bucket communicates with the parity buckets using the 1PC or the 2PC. The latter is as for an insert, except for the inverse result of the key c test. As for the insert for $k > 1$, the parity buckets keep also the Δ-record till the commit message. More generally, the degraded mode for a delete is analogous to that of an insert.

11.5 Merge

Deletions may decrease the number of records in a bucket under an optional threshold $b' \ll b$, e.g., 0.4 b. The bucket reports this to the coordinator. The coordinator may start a bucket merge operation. The merge removes the last data bucket in the file, provided the file has at least two data buckets. It moves the records in this bucket back to its parent bucket that has created it during its split. The operation increases the load of the file.

In the normal mode, for $n > 0$, the merge starts with setting the split pointer n to $n := n - 1$. For $n = 0$, it sets $n = 2^{i=1} - 1$. Next, it moves the data records of bucket $n + 2^i$ (the last in the file), back into bucket n (the parent bucket). There, each record gets a new rank, following consecutively the ranks of the records already in the bucket. The merge finally removes the last data bucket of the file that is now empty. For $n = 0$ and $i > 0$, it decreases i to $i = i - 1$.

If n is set to 0, the merge may also decrease K by one. This happens if N decreases to a value that previously caused K to increase. Since merges are rare, and merges that decrease K are even more rare, we omit discussion of the algorithm for this case.

The merge also updates the k parity buckets. This undoes the result of a split. The number of parity buckets in the bucket group can remain the same. If the removed data bucket was the only one in its group, then all the k parity buckets for this group are also deleted. The merge commits the parity updates using 1PC or 2PC. It does it similarly to what we have discussed for splits.

As for the other operations, the degraded mode for a merge starts when any of the buckets involved does not reply. The sender, other than the coordinator itself, alerts the latter. The various cases with which we encountered similar to those already discussed. Likewise, the 2PC termination algorithms in the degraded mode are similar to those for an insert or a delete. As for the split, every bucket involved reports any unavailability. We omit the details.

12. PERFORMANCE ANALYSIS

We analyze here formally the storage and messaging costs. We deal with the dominant factors only. This shows the typical performance of our scheme and its good behavior in practice. The storage overhead for parity data, in particular, appears about the best possible. We conclude with examples showing how to use the outcome for practical design choices.

12.1 Storage Occupancy

The *file load factor* α is the ratio of the number of data records in the file over the capacity of the file buckets. The average load factor α_d of the LH*$_{RS}$ data buckets is that of LH*. Under the typical assumptions (uniform hashing, few overflow records . . .), we have $\alpha_d = \ln(2) \approx 0.7$. Data records in LH*$_{RS}$ may be slightly larger than in LH*, since it may be convenient to store the rank with them.

The parity overhead should be about k/m in practice. This is the minimal possible overhead for k-available record or bucket group. Notice that parity records are slightly larger than data buckets since they contain additional fields. If we neglect these aspects, then the load factor of a bucket group is typically

$$\alpha_{g=}\alpha_d/(1+k/m).$$

The average load factor α_f of the file depends on its state. As long as the file availability level K' is the intended one K, we have $\alpha_f = \alpha_g$, provided $N \gg m$, so that the influence of the last group is negligible. The last group contains possibly less than m data buckets,. If $K' = K - 1$, that is, if the file is in the process of scaling to a higher availability level, then α_f depends on the split pointer n and file level i as follows:

$$\alpha_f \approx \alpha_d((2^i - n)/(1+(K-1)/m) + 2n/(1+K/m))/(2^i + n).$$

There are indeed $2n$ buckets in the groups with $k = K$ and $(2^i - n)$ bucket in the groups whose $k = K'$. Again, we neglect the possible impact of the last group. If $\alpha_g\ (k)$ denotes α_g for given k, we have

$$\alpha_g(K'+1) < \alpha_f < \alpha_g(K').$$

In other words, α_f is then slightly lower than $\alpha_g(K')$. It decreases progressively until it reaches its lower bound for K', reaching it for $n = 2^{i+1} - 1$. Then, if $n = 0$ again, K' increases to K, and α_f is $\alpha_g\ (K')$ again.

The increase in availability should in practice, concern, relatively few N values of an LH*$_{RS}$ file. The practical choice of N_1 should indeed be $N_1 \gg 1$. For any intended availability level K and of group size m, the load factor of the scaling LH*$_{RS}$ file should be therefore, in practice, about constant and equal to $\alpha_g\ (K)$. This is the highest possible load factor for the availability level K and α_d. We thus achieve the highest possible α_f for any technique added upon an LH* file to make it K-available.

Our file availability grows incrementally to level $K+1$. Among the data buckets, only those up to the last one, split and those newly created since the split pointer was reset to zero have this higher availability. This strategy induces a storage occupancy penalty with respect to best $\alpha_f(K)$ as long as the file does not reach the new level. The worst case for K-available LH*$_{RS}$ is then, in practice, $\alpha_f\ (K+1)$. This value is, in our case, still close to the best for $(K+1)$-available file. It does not seem possible to achieve a better evolution of α_f for our type of an incremental availability increase strategy.

The record group size m limits the record and bucket recovery times. If this time is of lesser concern than the storage occupancy, one can set m to a larger

value, for example, 64, 128, 256... Then, all k values needed in practice should remain negligible with respect to m, and $N \gg 1$. The parity overhead becomes negligible as well. The formula for α_f becomes $\alpha_f \approx \alpha_d/(1 + k/\min(N, m))$. It converges rapidly to α_d, while N scales up especially for the practical choices of N_i for the scalable availability. We obtain high availability at almost no storage occupancy cost.

Observe that for given α_f and the resulting acceptable parity storage overhead, the choice of a larger m benefits the availability. While choosing for an α_f some m_1 and k_1 leads to the k_1-available file, the choice of $m_2 = l m_1$ allows for $k_2 = l k_1$ which provides l more times available file. However, the obvious penalty is about l times greater messaging cost for bucket recovery since m buckets have to be read. Fortunately, this does not mean that the recovery time also increases l times as we will see. Hence, the trade-off can be worthy in practice.

Example 10. We now illustrate the practical consequences of the above analysis. Assume $m = 8$. The parity overhead is then (only) about 12.5% for the 1-availability of the group, 25% for its 2-availability and so on.

We also choose uncontrolled scalable availability with $N_1 = 16$. We thus have 1-available file, up to $N = 16$ buckets. We can expect $\alpha_f = \alpha_g(1) \approx 0.62$ which is the best for this availability level, given the load factor α_d of the data buckets. When $N := 16$, we set $K := 2$. The file remains still only 1-available until it scales to $N = 32$ buckets. In the meantime, α_f decreases monotonically to ≈ 0.56. At $N = 32$, K' reaches K, and the file becomes 2-available. Then, α_f becomes again the best for the availability level and remains so until the file reaches $N = 256$. It stays optimal for a fourteen times longer period than when the availability transition was in progress and the file load was below the optimal one of $\alpha_g(1)$. Then, we have $K := 3$, and so both.

Assume now a file that has currently $N = 32$ buckets and is growing up to $N = 256$, hence it is 2-available. The file tolerates the unavailability of buckets 8 and 9 and, separately, that of bucket 10. However, unavailability of buckets 8–10 is catastrophic. Consider then rather the choice of $m = N_1 = 16$ for the file starting with $K = 2$. The storage overhead remains the same, hence is α_f. Now, the file tolerates that unavailability as well, even that of up to any four buckets among 1 to 16.

Consider further the choice of $m = 256$ and of $N_1 = 8$. Then, $K' = 1$ until $N = 16$, $K' = 2$ until $N = 128$, then $K' = 3$ and so on. For $N = 8$, $\alpha_f = \alpha_d/(1^1/_8) \approx 0.62$. For $N = 9$, it drops to $\alpha_d/(1^1/_4) \approx 0.56$. It increases monotonically again to $\alpha_f \alpha_d/(1^1/_8)$ for $N16$ when the file becomes 2-available. Next, it continues to increase towards $\alpha_f = \alpha_d/(1^2/_{64}) \approx 0.68$ for $N = 64$. For $N = 65$, it decreases again to $\alpha_f = \alpha_d/(1^3/_{64}) \approx 0.67$. Next, it increases back to 0.68 for $N = 128$ when the file becomes 3-available. It continues towards almost 0.7 when N scales. And so on, with α_f about constantly equal to almost 0.7 for all practical file sizes. The file has to reach $N = 2^{3k+1}$ buckets to become k-available. For instance, it has to scale to a quite sizable 32M buckets to reach $k = 8$. The file still keeps the parity overhead k/m, rather negligible since it is under 3%.

Table VI. Messaging Costs of an LH^*_{RS} File

Manipulation	Normal Mode (N)	Degraded Mode (D)
Bucket Recovery (B)	$B \approx (3 + 2m + 3k) + \alpha_d\, bm + \alpha_d\, b(l-1) + 1$	Not Applicable
Record Rec. (R)	$R \approx 2$ or $2(m-1)$	Not Applicable
Search (S)	$S_N \approx 2$	$S_D \approx S_N + R$
Insert (I)	$I_N \approx 4$ or $2 + 3k$	$I_D \approx 1 + I_N + B$
Delete (D)	$D_N \approx 2$ or $1 + 3k$	$D_D \approx 1 + D_N + B$
Scan (C)	$C_N \approx 1 + N$	$C_D \approx C_N + l(1 + B_1)$
Split (L)	$L_N \approx 1 + 0.5\alpha_d\, b(2I_N = 1)$	$L_D \approx L_N + B$
Merge (M)	$M_N = L_N$	$M_D \approx M_N + B$

12.2 Messaging

We calculate the messaging cost of a record manipulation as the number of (logical) messages exchanged between the SDDS clients and servers to accomplish the operation. This performance measure has the advantage of being independent of various practical factors such as network, CPU performance, communication protocol, flow control strategy, bulk messaging policy and so forth. We consider one message per-record sent or received, or a request for a service, or a reply carrying no record. We assume reliable messaging. In particular, we consider that the network level handles message acknowledgments unless this is part of the SDDS layer, for example, for the synchronous update of the parity buckets. The sender considers a node unavailable if it cannot deliver its message.

Table VI shows the typical messaging costs of an LH^*_{RS} file operation for both normal and degraded mode. The expressions for the latter may refer to the costs for the normal mode. We present the formulas for the dominant cost component. Their derivation is quite easy, hence we only give an overview. More in depth formulas such as those for average costs, seem difficult to derive. Their analysis remains an open issue. Notice however that the analysis of the messaging costs for LH^* in [Litwin et al. 1996] applies to the messaging costs of LH^*_{RS} data buckets alone in normal mode.

To evaluate bucket recovery cost in this way, we follow the scheme in Section 4.1. A client encountering an unavailable bucket sends a message to the coordinator. The coordinator responds by scanning the bucket group, receiving acknowledgments of survivors, selecting spares, receiving acknowledgments from them, and selecting the recovery manager. This gives us a maximum of $3 + 2m + 3l$ setup messages (if l buckets have failed). Next, the recovery manager reads m buckets filled on average with αb records each. It dispatches the result to $l-1$ spares, using one message per-record, since we assume reliable delivery. We also assume that, typically, the coordinator finds only the unavailable data buckets. Otherwise the recovery cost is higher as we recover parity buckets in 2nd step, reading the m data buckets. Finally, the recovery manager informs the coordinator.

For record recovery, the coordinator forwards the client request to an unavailable parity bucket. That looks for the rank of the record. If the record does not exist, two messages follow, to the coordinator and to the client. Otherwise, $2(m-1)$ messages are typically, and at most, necessary to recover the record.

The other costs formulas are straightforward. The formulas for the insert and delete consider the use of 1PC or 2PC. We do not provide the formulas for the updates. The cost of a blind update is that of an insert. The cost of a conditional update is that of a key search, plus the cost of the blind one. Notice however that because of the specific 2PC, the messages of an update to $k > 1$ parity buckets are sequential. The values of S_N, I_N, D_N, and C_N do not consider any forwarding to reach the correct bucket. The calculus of C_N considers the propagation by multicast. We also assume that l unavailable buckets found are each in a different group. The coefficient B_1 denotes the recovery cost of a single bucket. Several formulas can obviously be simplified without noticeable loss of precision in practice. Some factors should be typically largely dominant, the B costs especially.

The parity management does not impact the normal search costs. In contrast, the parity overhead of the normal updating operations is substantial. For $k = 1$, it doubles I_N and D_N costs with respect to those of LH*. For $k > 1$, it is substantially more than the costs for manipulating the data buckets alone as in LH*. Already for $k = 2$, it implies $I_N = D_N = 8$. Each time we increment k, an insert or delete incurs three more messages.

The parity overhead is similarly substantial for split and merge operations as it depends on I_N. The overhead of related updates is linearly dependent on k. Through k and the scalable availability strategy, it is also indirectly dependant on N. For the uncontrolled availability, the dependence is of the order of $O(\log_{N1} N)$. A rather large N_1 should suffice in practice, usually, at least $N_1 = 16$. Thus this dependence should little affect the scalability of the file.

The messaging costs of recovery operations are linearly dependent on m and l. The bucket recovery also depends linearly on b. While increasing m benefits α_f, it proportionally affects the recovery. To offset the incidence at B, one may possibly decrease b accordingly. This increases C_N for the same records, since N increases accordingly. This does not mean, however, that the scan time increases as well. In practice, it should often decrease.

13. VARIANTS

There are several ways to enhance the basic scheme with additional capabilities or to amend the design choices so as to favor specific capabilities at the expense of others. We now discuss a few such variations, potentially attractive to some applications. We show the advantages, but also the price to pay for them, with respect to the basic scheme. First, we address the messaging of the parity records. Next, we discuss on-demand tuning of the availability level and of the group size. We also discuss a variant where the data bucket nodes share the load of the parity records. We recall that in the basic scheme, the parity and data records are at separate nodes. The sharing substantially decreases the total number of nodes necessary for a larger file. Finally, we consider alternative coding schemes.

13.1 Parity Messaging

Often an update changes only a small part of the existing data record. For instance, this is the case with a relational database where an update concerns

usually one or a few attributes among many. For such applications, the Δ-record would consist mainly of zeros except for a few symbols. If we compress the Δ-record and no longer have to transmit these zeroes explicitly, our messages should be noticeably smaller.

Furthermore, in the basic scheme, the data bucket manages its messaging to every parity bucket. It also manages the rank that it sends along with the Δ-record. An alternative design sends the Δ-record only to the first parity bucket, and without a rank. The first parity bucket assigns the rank. It is also in charge of the updates to the $k - 1$ other parity buckets, if there are any, using 1PC or 2PC. The drawback of the variant design is that updating needs two rounds of messages; the advantage is simpler parity management at the data buckets. The 1PC suffices for the dialog between the data bucket and the first parity bucket. The management of the ranks also becomes transparent to the data buckets, as well as the scalable availability. The parity subsystem is more autonomous. An arbitrary 0-available SDDS scheme can be more easily generalized to a highly-available scheme.

Finally, it is also possible to avoid the commit ordering during 2PC for updates. It suffices to add to each parity record the *commit state* field which we call S. The field has the binary value s_l per lth data bucket in the group. When a parity bucket p gets the commit message from this bucket, it sets s_l to $s_l = s_l$ XOR 1. If bucket p alerts the coordinator because of lack of a commit message, the coordinator probes all other available parity buckets for their s_l. The parity update is done if and only if any bucket p' probed had $s_l^{p'} \neq s_l^p$. Recall that the update is posted to all or none of the available parity buckets that are not in the ready-to-commit state during the probing. The coordinator synchronizes the parity buckets accordingly, using the Δ-record in the differential file of bucket p. The advantage is a faster commit process since the data bucket may send messages in parallel. The disadvantage is an additional field to manage, only necessary for updates.

13.2 Availability Tuning

We can add to the basic data record manipulations the operations over the parity management. First, we may wish to be able to decrease or increase the availability level K of the file. Such *availability tuning* could perhaps reflect past experience. It differs from scalable availability, where splits change k incrementally. To decrease K, we drop, in one operation, the last parity bucket(s) of every bucket group. Vice versa, to increase the availability, we add the parity bucket(s) and records to every group. The parity overhead decreases or increases accordingly as well as the cost of updates.

More precisely, to decrease the availability of a group from $k > 1$ to $k - 1$, it suffices to delete the kth parity bucket in the group. The parity records in the remaining buckets do not need to be recomputed. Notice that this is not true for some of the alternative coding schemes we discuss in the following. This reorganization may be trivially set up in parallel for the entire file. As the client might not have all the data buckets in its image, it may use as the

basis the scan operation discussed previously. Alternatively, it may simply send the query to the coordinator. The need being rare, there is no danger of a hot spot.

To add a parity bucket to a group requires a new node for it, with $(k + 1)$ column of $\mathbf{Q}$ (or $\mathbf{P}$). Next, one should read all the data records in the group and calculate the new parity records as if each data record was an insert. Various strategies exist to efficiently read data buckets in parallel. Their efficiency remains for further study. As noted previously, it is easy to set up the operation in parallel for all the groups in the file. Also, as noted, the existing parity records do not need the recalculation, unlike other candidate coding schemes for LH*RS that we investigate later.

Adding a parity bucket operation can be concurrent with normal data bucket updates. Some synchronization is, however, necessary for the new bucket. For instance, the data buckets may be made aware of the existence of this bucket before it requests the first data records. As a result, they will start sending the Δ-record for each update coming afterwards. Next, the new bucket may create its parity records in their rank order. Then the bucket encodes any incoming Δ-record it did not request. This, provided it already has created the parity record, hence it processed its rank. It disregards any other Δ-record. In both cases, it commits the Δ-record. The parity record will include the disregarded Δ-record when the bucket will encode the data records with that rank, then also requesting the Δ-record.

13.3 Group Size Tuning

We recall that the group size m for LH*RS is a power of two. The group size tuning may double or halve m synchronously for the entire file one or more times. The doubling merges two successive groups, which we will call left and right, that become a single group of $2m$ buckets. The first left-group starts with bucket 0. Typically the merged groups each have k parity buckets. Seldom the right group will have an availability level of $k - 1$ when the split pointer is in the left group and the file is changing its availability level. We discuss the former case only. The generalization to the latter and to the entire file is trivial.

The operation reuses the k buckets of the left group as the parity buckets for the new group. Each of the $k - 1$ columns of the parity matrices $\mathbf{P}$ and $\mathbf{Q}$ for the parity buckets other than the first one is, however, now provided with $2m$ elements instead of top m only as previously. The parity for the new group is computed in these buckets as if all the data records in the right group were reinserted to the file. There are a number of ways to perform this operation efficiently that remain for further study. It is easy to see, however, that, for the first new parity bucket, a faster computation may consist simply in XORing rank-wise the B-field of each record with that of the parity record in the first bucket of the right group, and unioning their key lists. Once the merge ends, the former parity buckets of the right group are discarded.

In contrast, the group size halving splits each group into two. The existing k parity buckets become those of the new left group. The right group gets k

new empty parity buckets. In both sets of parity buckets, the columns of **P** or **Q** need only the top m elements. Afterwards, each record of the right group is read. It is then encoded into the existing buckets as if it was deleted, that is, its key is removed from the key list of its parity records and its nonkey data are XORed to the B-fields of these records. At in the same time, it is encoded into the new parity buckets as if it was just inserted into the file. Again, there are a number of ways to implement the group size halving efficiently that remain open for study.

13.4 Parity Load Balancing

In the basic scheme, the data and parity buckets are at separate nodes. A parity bucket also sustains the updating processing load up to m times that of the update load of a data bucket as all the data buckets in the group may get updated simultaneously. The scheme requires about Nk/m nodes for the parity buckets, in addition to N data bucket nodes. This number scales out with the file. In practice, for a larger file, for example, on $N = 1K$ data nodes with $m = 16$ and $K = 2$, this leads to 128 parity nodes. These parity nodes do not carry any load for queries. On the other hand, the update load on a parity bucket is about 16 times that of a data bucket. If there are intensive burst of updates, the parity nodes could form a bottleneck that slows down commits. This argues against using larger m. Besides, some user may be troubled with the sheer number of additional nodes.

The following variant decreases the storage and processing load of the parity records on the nodes supporting them. This happens provided that $k \leq m$ which seems a practical assumption. It also balances the load so that the parity records are located mostly on data bucket nodes. This reduces the number of additional nodes needed for the parity records to m at most. The variant works as follows.

Consider the ith parity record in the record group with rank r, $i = 0, 1 \ldots k-1$. Assume that for each (data) bucket group, there is a parity bucket group of m buckets, numbered $0, 1 \ldots m-1$, of capacity kb/m records each. Store each parity record in parity bucket $j = (r+i) \bmod m$. Do it as the primary record, or an overflow one if needed, as usual. Place the m parity buckets of the first group, that is, containing data buckets $0, \ldots, m-1$, on the nodes of the data buckets of its immediately right group, that is, with data buckets $m, \ldots, 2m-1$. Place the parity records of this group on the nodes of its (immediately) left group. Repeat for any next groups while the file scales out.

The result is that each parity record of a record group is in a different parity bucket. Thus, if we no longer can access a parity bucket, then we lose access to a single parity record per group. This is the key requirement to the k-availability for the basic scheme. The LH^*_{RS} file remains, consequently, K available. The parity storage overhead, that is, the parity bucket size at a node, decreases now uniformly by factor m/k. In our example, it divides by 8. The update load on a parity bucket also becomes twice that of a data bucket. In general, the total processing and storage load is about balanced over the data nodes for both the updates and searches.

The file needs, at most, m additional nodes for the parity records. This happens when the last group is the left one and the last file bucket $N-1$ is its last one. When this bucket is the last in a right group, the overhead is zero. On average over N, the file needs $m/2$ additional nodes. The number of additional nodes becomes a constant and a parameter, independent of the file size. The total number of nodes for the file becomes $N+m$ at worst. For a larger file, the difference with respect to the basic scheme is substantial. In our example, the number of additional nodes drops from 128 to 8 at most and 4 on average. In other words, it reduces from 12.5% to less than 1% at worst. For our $N = 1K$, it drops, in fact, to zero since the last group is the right one. The file remains 2-available.

Partitioning should usually also shrink the recovery time. The recovery operation can now occur in parallel at each parity bucket. The time for decoding the data records in the l unavailable data buckets is then close to l/m fraction of the basic one. In our previous example, the time to decode a double unavailability decreases accordingly 8 times. The total recovery time would not decrease that much. There are other phases of the recovery process whose time remains basically unchanged. The available data records still have to be sent to the buckets performing the operation, the decoded records have to be sent to the spare and inserted there, and so on. A deeper design and performance analysis of the scheme remain to be done.

Notice finally, that if $n > 1$ nodes, possibly spares may participate in the recovery calculus, then the idea described above previously of partitioning a parity bucket onto the n nodes may be usefully applied to speed up the recovery phase. The partitioning would become dynamically the first step of the recovery process. As discussed, this would decrease the calculus time by a factor possibly reaching l/n. The overall recovery time may improve as well. The gain may be substantial for large buckets and $n \gg 1$.

13.5 Alternative Erasure Correcting Codes

In principle, we can retain the basic LH*RS architecture with a different erasure correcting code. The interest in these codes stems first from the interest in higher availability RAID [Hellerstein et al. 1994; Schwarz and Burkhard 1996; Burkhard and Menon 1993; Blaum et al. 1993; Blaum et al. 1995; Blomer et al. 1995; Schwabe and Sutherland 1996]. Proposals for high-availability storage systems [Cooley et al. 2003; Zin et al. 2003] (encompassing thousands of disks for applications such as large email servers [Manasse 2002]), massive downloads over the World Wide Web [Byers et al. 1998; Byers et al. 1999], and globally distributed storage [Anderson and Kubiatowicz 2002; Weather-Spoon and Kubiatowicz 2002] maintain constant interest in new erasure correcting codes. These may compare favorably with generalized Reed-Solomon codes. (In addition, decoding Reed-Solomon codes has made great strides [Sudan 1997; Guruswami and Sudan 1999], but we use them only for erasure correction for which classic linear algebra seems best.) Nevertheless, one has to be careful to carry over the conclusions about the fitness of a code to LH*RS. Our scheme is, indeed, largely different from these applications. It favors smaller group sizes (to limit communication costs during recovery), utilizes main memory (hence

is sensitive to parity overhead), can recover small units (the individual record), has scalable availability, and so forth.

We will now discuss replacing our code with other erasure correcting codes within the scope of our scheme. Certain codes allow a trade-off with performance factors. Typically, a variant can offer faster calculus than our scheme at the expense of parity storage overhead or limitations on the maximum value of k. For the sake of comparison, we first list a number of necessary and desirable properties for a code. Next, we discuss how our code fits them. Finally, we use the framework for the analysis.

The design Properties for an Erasure Correcting Code for LH*$_{RS}$ include the following:

(1) Systematic code. The code words consist of data symbols concatenated with parity symbols. This means that the application data remains unchanged and that the parity symbols are stored separately.
(2) Linear code. We can use Δ-records when we update, insert or delete a single data record. Otherwise, after a change, we would have to access all data records and recalculate all parity from them.
(3) Minimal, or near-minimal, parity storage overhead.
(4) Fast encoding and decoding.
(5) Constant bucket group size, independent of the availability level.

Notice that it is property (2) that also allows us to compress the delta record by only transmitting nonzero symbols and their location within the delta record.

Our codes (as defined in Section 3) fulfill all these properties. They are systematic and linear. They have minimal possible overhead for parity data within a group of any size. This is a consequence of being Maximum Distance Separable (MDS). Since the parity matrix contains a column of ones, record reconstruction in the most important case (a single data record unavailability) proceeds at the highest speed possible. As long as $k = 1$, any update incurs the minimal parity update cost for the same reason. In addition, for any k, updates to a group's first data bucket will also result in XORing because of the row of ones in the parity matrix. Finally, we can use the logarithmic matrices.

Our performance results (Section 5) show that the update performance at the second, third, and so on, parity buckets is adequate. We recall that, for $GF(2^{16})$, the slow down was 10% for the 2nd parity bucket and an additional 7 % for the 3rd one, with respect to the 1st bucket only (Table III). It is impossible to improve the parity matrix further by introducing additional one-coefficients to avoid *GF* multiplication (we omit the proof of this statement). Next, a bucket group can be extended to a total of $n = 257$ or $n = 65,537$, depending upon whether we use the Galois field with 2^8 or 2^{16} elements. Up to these bounds, we can freely choose m and k, subject to $m + k = n$, in particular, we can keep m constant. An additional nice property is that small changes in a data record result in small changes in the parity records. In particular, if a single bit is changed, then a single parity symbol only in each parity record changes (except for the first parity record where only a single bit changes).

					a_0
					a_1
				a_0	$a_2 \oplus b_0$
				$a_1 \oplus b_0$	$a_3 \oplus b_1$
a_0	b_0	c_0	$a_0 \oplus b_0 \oplus c_0$	$a_2 \oplus b_1 \oplus c_0$	$a_4 \oplus b_2 \oplus c_0$
a_1	b_1	c_1	$a_1 \oplus b_1 \oplus c_1$	$a_3 \oplus b_2 \oplus c_1$	$a_5 \oplus b_3 \oplus c_1$
a_2	b_2	c_2	$a_2 \oplus b_2 \oplus c_2$	$a_4 \oplus b_3 \oplus c_2$	$a_6 \oplus b_4 \oplus c_2$
a_3	b_3	c_3	$a_3 \oplus b_3 \oplus c_3$	$a_5 \oplus b_4 \oplus c_3$	$a_7 \oplus b_5 \oplus c_3$
a_4	b_4	c_4	$a_4 \oplus b_4 \oplus c_4$	$a_6 \oplus b_5 \oplus c_4$	$a_8 \oplus b_6 \oplus c_4$
a_5	b_5	c_5	$a_5 \oplus b_5 \oplus c_5$	$a_7 \oplus b_6 \oplus c_5$	$a_9 \oplus b_7 \oplus c_5$

Fig. 15. Convolutional array code.

Candidate codes are two-dimensional codes in which the parity symbols are the XOR of symbols in lines in one or more directions. One type is the *convolutional array* codes that we discuss now. We address some others later in this section. The convolutional codes were developed originally for tapes, adding parity tracks with parity records to the data tracks with data records [Prusinkiewicz and Budkowski 1976; Patel 1985; Fuja et al. 1986]. Figure 15 shows an example with $m = 3$ data records and $k = 3$ parity records. The data records form the three left-most columns, that is, $a_0, a_1, \ldots, b_0, b_1, \ldots, c_0, c_1, \ldots$ Data record symbols with indices higher than the length of the data record are zero; in our figure, this applies to a_6, a_7, and so on. The next three columns numbered $K = 0, 1, 2$ contain the parity records. The record in parity column 0 contains the XOR of the data records along a line of *slope* 0, that is, a horizontal line. Parity column 1 contains the XOR of data record symbols aligned in a line of slope 1. The final column contains the XOR along a line of slope 2.

The last two columns are longer than the data columns. They have an overhang of 2 and 4 symbols, respectively. In general, parity record or column K has an overhang of $K(m-1)$ symbols. A group with k parity records and m data records of length L has a combined overhang of $k(k-1)(m-1)/2$ symbols so that the parity overhead comes to $\frac{k}{m} + \frac{k(k-1)(m-1)}{2mL}$ symbols. The first addend is the minimal storage overhead of any MDS code. The second addend shows that a convolutional array code with $k > 1$ is not MDS. The difference is, however, typically not significant. For instance, choosing $k = 5$, $m = 4$, and $L = 100$ (the record length in our experiments) adds only 5%.

The attractive property of a convolutional code in our context is its updating and decoding speed. During an update, we change all parity records only by XORing them with the Δ-record. We start at different positions in each parity record, see Figure 15. The updates proceed at the fastest possible speed for all data and parity buckets. Unlike our case, this is true only for the first parity bucket and the first data bucket. Likewise, the decoding iterates by XORing and shifting of records. This should be faster than our *GF* multiplications. Notice,

a_0	b_0	(c_0)	$a_0 \oplus a_1 \oplus b_0 \oplus b_2 \oplus c_0 \oplus c_3$	$a_1 \oplus b_2 \oplus c_3$
a_1	b_1	c_1	$(a_0 \oplus a_2 \oplus b_0 \oplus b_1 \oplus b_2 \oplus b_3 \oplus c_0 \oplus c_1 \oplus c_3)$	$a_0 \oplus a_1 \oplus a_2 \oplus b_0 \oplus b_2 \oplus b_3 \oplus c_0 \oplus c_3$
a_2	b_2	c_2	$a_0 \oplus a_3 \oplus b_0 \oplus b_1 \oplus b_3 \oplus c_1 \oplus c_2 \oplus c_3$	$(a_0 \oplus a_2 \oplus a_3 \oplus b_0 \oplus b_1 \oplus b_2 \oplus b_3 \oplus c_1 \oplus c_3)$
(a_3)	b_3	c_3	$a_0 \oplus b_1 \oplus c_2$	$a_0 \oplus a_3 \oplus b_1 \oplus b_3 \oplus c_2 \oplus c_3$
0	(0)	0	0	0

Fig. 16. A codeword of $B_2(5)$ with a fictional row of zeroes added.

however, that writing a generic decoding algorithm for any m and k is more difficult than for the RS code.

All things considered, these codes can replace RS codes in the LH^*_{RS} framework, offering faster performance at the costs of larger parity overhead. Notice that we can also reduce the parity overhead by using negative slopes at the added expense of the decoding complexity (inversion of a matrix in the field of Laurent series over $GF(2)$).

Block array codes are another type of codes that are MDS. They avoid the overhang in the parity records. As an example, we sketch the code family $B_k(p)$, [Blaum et al. 1998], where k is the availability level and p is a prime, corresponding to our m, that is, $p \geq k + m$. Prime p is not a restriction since we may introduce dummy symbols and data records.

In Figure 15 for instance, a_i, b_i, c_i with $i > 5$ are dummy symbols. Next, in Figure 16, we have chosen $k = 2$ and $m = 3$, hence $p = 5$. We encode first four symbols from three data records $a_0, a_1, \ldots$, $b_0, b_1, \ldots$, and $c_0, c_1, \ldots$ The pattern repeats, following symbols in groups of four symbols. We arrange the data and parity records as the columns of a 4 by 5 matrix. For ease of presentation and because slopes are generally defined for square matrices, we added a fictional row of zeroes (which are not stored). We now require that the five symbols in all rows and all lines of slope -1 in the resulting 5 by 5 matrix have parity zero. The line in parentheses in Figure 15 is the third such line.

Block array codes are linear and systematic. As for our code, we update the parity records using Δ-records. As the figure illustrates, we only use XORing. In contrast to our code and to the convolutional array code, the calculus of most parity symbols involves more than one Δ-record symbol. For example, the updating of the 1st parity symbol in Figure 16 requires XORing of two symbols of any Δ-record, for instance, the first and second symbol of the Δ-record if record $a_0, a_1, \ldots$ changes. This results in between one and two times more XORing. Decoding turns out to have about the same complexity as encoding for $k = 2$. All this should translate into faster processing than for our code.

For $k \geq 3$, we generalize by using k parity columns, increasing p if needed, and requiring parity zero along additional slopes $-2, -3$, and so forth. In our example, increasing k to 3 involves setting p to the next prime, which is 7, to accommodate the additional parity column and adding a dummy data record to each record group. We could use $p = 7$ also for $k = 2$, but this choice slows down the encoding by adding terms to XOR in the parity expressions. The main problem with $B_k(p)$ for $k > 2$ is that the decoding algorithm becomes fundamentally more complicated than for $k = 2$. Judging from the available literature, an implementation is not trivial, and we can guess that even an optimized decoder should perform slower than our RS decoder, [Blaum

et al. 1998]. All things considered, using $B_k(p)$ does not seem a good choice for $k > 2$.

The *EvenOdd* code, [Blaum et al. 1993, 1995, 1998], is a variant of $B_2(p)$ that improves encoding and decoding. The idea is that the 1st parity column is the usual parity, and the 2nd parity column is either the parity or its binary complement of all the diagonals of the data columns with the exception of a special diagonal whose parity decides on the alternative to be used. The experimental analysis in Schwarz [2003] showed that both encoding and decoding of EvenOdd are faster than for our fastest RS code. In the experiment, EvenOdd repaired a double record erasure four times faster. The experiment did not measure the network delay so that the actual performance advantage is less pronounced. It is, therefore, attractive to consider a variant of LH*RS using EvenOdd for $k = 2$. An alternative to EvenOdd is the *Row-Diagonal Parity* code presented in Corbett et al. [2004].

EvenOdd can be generalized to $k > 2$, [Blaum et al. 1998]. For $k = 3$, one obtains an MDS code with the same difficulties of decoding as for $B_3(p)$. For $k > 3$, the result is known to not be MDS.

A final block-array code for $k = 2$ is *X-code* [Xu and Bruck 1999]. These have zero parity only along the lines with slopes 1 and -1 and, as all block-array codes, use only XORing for encoding and decoding. They too seem to be faster than our code, but they cannot be generalized to higher values of k.

*Low density parity check (*LDPC*) codes* are systematic and linear codes that use a large $M \times N$ parity matrix with only zero and one coefficients [Alon et al. 1995; Berrou and Glavieux 1996; Gallager 1963; Mackay and Neal 1997, 1999; Mackay 2000]. We can use bits, bytes, words, or larger bit strings as symbols. The *weights*, that is the number of ones in a parity matrix column or row are always small, and often a constant. Recent research (e.g., Luby [1997], [Cooley et al. 2003]) established the advantage of varying weights. We obtain the parity symbols by multiplying the M-dimensional vector of data units with the parity matrix. Thus, we generate a parity symbol by XORing w data units, where w denotes the column weight.

Good LDPC codes use sparse matrix computation to calculate most parity symbols, resulting in fast encoding. Fast decoders exist as well [Mackay 2000]. LDPC codes are not MDS, but good ones come close to being MDS. Speed and closeness to MDS improve as the matrix size increases. The Digital Fountain project used a Tornado (LDPC) code for $M = 16000$, with 11% additional storage overhead at most [Byers et al. 1998]. Mackay [2000] gives a very fast decoding LDPC with $M = 10,000$.

There are several ways to apply sparse matrix codes to LH*RS. One is to choose the byte as the data unit size and use chunks of M/m bytes per data bucket. Each block of M bytes is distributed so the ith chunk is in ith bucket. Successive chunks in a bucket come from successive blocks. The number of chunks and their size determines the bucket length.

Currently, the best M values are large. A larger choice of m increases the load on the parity record during the updates similar to the record groups using our coding. This choice also increases recovery cost. However, the choice of the m value is less critical as there is no m by m matrix inversion. Practical values

of m appear to be $m = 4, 8, 16, 32$. If the application record is in Kbytes, then a larger m allows for a few chunks per record or a single one. If the record size is not a chunk multiple, then we pad the last bytes with zeros. One can use Δ-records calculated over the chunk(s) of the updated data record to send updates to the parity buckets since LDPC codes are linear.

If application data records consist of hundreds of bytes or smaller, then it seems best to pack several records into a chunk. As typical updates address only a single record at a time, we should use compressed Δ-records. Unlike our code, however, an update will usually change more parity symbols than in the compressed Δ-record. This obviously comparatively affects the encoding speed.

In both cases, the parity records would consists of full parity chunks of size $M/m+\varepsilon$, where ε reflects the deviation from MDS, for example, the 11% quoted previously. The padding, if any, introduces some additional overhead. The incidence of all the discussed details on the performance of the LDPC coding within LH^*_{RS} as well as further related design issues are open research problems. At this stage, all things considered, the attractiveness of LDPC codes is their encoding and decoding speed which is close to the fastest possible, that is, of the symbol-wise XORing of the data and parity symbols, like the first parity record of our coding scheme [Byers et al. 1999]. Notice, however, that encoding and decoding are only part of the processing cost in LH^*_{RS} parity management. The figures in Section 5 show that the difference in processing using only the first parity bucket and the others is not that pronounced. Thus, the speed-up resulting from replacing RS with a potentially faster code is limited. Notice also that finding good LDPC codes for smaller M is an active research area.

RAID Codes. The interest in RAID generated specialized erasure correcting codes. One approach is XOR operations only, generating parity data for a k-available disk array with typical values of $k = 2$ and $k = 3$, for example, Hellerstein et al. [1994], Chee and Colbourn [2000], Cohen and Colbourn [2001]. For a larger k, the only RAID code known to us is based on the k-dimensional cube. RAID codes are designed for a relatively large number of disks, for example, more than 20 in the array. Each time we scale from $k = 2$ to $k = 3$ and beyond, we change the number of data disks. Implementing these changing group sizes would destroy the LH^*_{RS} architecture but could result in some interesting scalable availability variant of LH^*.

For the sake of completeness, we finally mention other flavors of generalized RS codes used in erasure correction, but not suited for LH^*_{RS}. The Digital Fountain project used a nonsystematic RS code in order to speed up the matrix inversion during decoding. The Kohinoor project developed a specialized RS code for group size $n = 257$ and $k = 3$ for a large disk array to support an email server. Plank [1997] seemed to give a simpler and longer (and hence better) generator matrix for an RS code, but Plank and Ding [2003] retracts this statement.

REFERENCES FOR APPENDIX C

ALON, N., EDMONDS, J., AND LUBY, M. 1995. Linear time erasure codes with nearly Optimal Recovery. In *Proceedings of the 36th Symposium on Foundations of Computer Science (FOCS)*.

BLOMER, J., KALFANE, M., KARP, R., KARPINSKI, M., LUBY, M., AND ZUCKERMAN, D. 1995. An XOR-based erasure-resilient coding scheme, Tech. rep., International Computer Science Institute, University of California, Berkeley, CA.

BLAUM, M., BRUCK, J., AND MENON, J. 1993. Evenodd: An optimal scheme for tolerating double disk failures in RAID architectures. *IBM Comput. Sci. Resear. Rep*. 11.

BLAUM, M., BRADY, J., BRUCK, J., AND MENON, J. 1995. EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures: *IEEE Trans. Comput. 44*, 192–202.

BLAUM, M., FARRELL, P. G., AND VAN TILBORG, H. 1998. Array codes, In *Handbook of Coding Theory*, vol. 2, Pless, V. S., Huffman, W. C. Eds. Elsevier Science.

BERROU, C. AND GLAVIEUX, A. 1996. Near optimum error correcting coding and decoding: Turbo-codes. *IEEE Trans. Comm. 44*, 1064–1070.

BYERS, J., LUBY, M., MITZENMACHER, M., AND REGE, M. 1998. A digital fountain approach to reliable distribution of Bulk Data. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (*SIGCOMM'98*) Vancouver, B.C. 56–68.

BYERS, J., LUBY, M., AND MITZENMACHER, M. 1999. Accessing multiple sites in parallel: Using tornado codes to speed up downloads. In *Proceedings of IEEE INFOCOM*, 275–283.

CHEE, Y., COLBOURN, C., AND LING, A. 2000. Asymptotically optimal erasure-resilient codes for large disk arrays. *Disc. Appl. Math. 102*, 3–36.

COHEN, M. C. AND COLBOURN, C. 2001. Optimal and pessimal orderings of Steiner triple systems in disk arrays. *Lat. Ameri. Theor. Inform*., 95–104.

COOLEY, J., MINEWEASER, J., SERVI, L., AND TSUNG, E. 2003. Software-based erasure codes for scalable distributed storage. In *USENIX Proceedings of the Conference on File and Storage Technologies* (*FAST'03*).

CORBETT, P., ENGLISH, B., GOEL, A., GRCANAC, T., KLEIMAN, S. LEONG, J., AND SANKAR, S. 2004. Row-diagonal parity for double disk failure correction. In *the 3rd Usenix Conference on File and Storage Technologies* (*FAST'04*). San Francisco, 1–14.

FUJA, T., HEEGARD, C., AND BLAUM, M. 1986. Cross parity check convolutional codes. *IEEE Trans. Inf. Theory 35*, 1264–1276.

GALLAGER, R. G. 1963. *Low-Density Parity Check Codes*. Research Monograph Series, MIT Press, Cambridge, MA.

GURUSWAMI, V. AND SUDAN, M. 1999. Improved decoding of Reed-Solomon and algebraic geometry codes. *IEEE Trans. Inf. Theory*, *45*, 6, 1757–1767.

MACKAY, D. J. C. AND NEAL, R. M. 1997. Near Shannon limit performance of low density parity check codes. *Electronic Letters 33*, 6, 457–458.

MACKAY, D. J. C. AND NEAL, R. M. 1999. Good error correcting codes based on very sparse matrices. *IEEE Trans. Inf. Theory 45*, 2, 399–431. Available at http://wol.ra.phy.cam.ac.uk/mackay.

MACKAY, D. 2000. Available at http://www.inference.phy.cam.ac.uk/mackay/CodesFiles.

MANASSE, M. 2002. Simplified construction of erasure codes for three or fewer erasures. Tech. rep. Kohinoor Project, Microsoft Research. Available at http://research.microsoft.com/research/sv/Kohinoor/ReedSolomon3.htm.

PATEL, A. M. 1985. Adaptive cross parity code for a high density magnetic tape subsystem. *IBM J. Resear. Develop*. *29*, 546–562.

PLANK, J. 1997. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Softw.—Prac. Exper*. *27*, 9, 995–1012.

PLANK, J. AND DING, Y. 2003. Correction to the 1997 tutorial on Reed-Solomon coding. Tech. rep., UT-CS-03-504, University of Tennessee.

PRUSINKIEWICZ, P. AND BUDKOWSKI, S. 1976. A double-track error-correction code for magnetic tape. *IEEE Trans. Comput. 19*, 642–645.

SCHWABE, E. J. AND SUTHERLAND, I. M. 1996. Flexible usage of redundancy in disk arrays. In *the 8th ACM Symposium on Parallel Algorithms and Architectures*, 99–108.

SCHWARZ, T. AND BURKHARD, W. 1996. Reliability and performance of RAIDs. *IEEE Trans. Magnetics 31*, 1161–1166.

SUDAN, M. 1997. Decoding of Reed Solomon codes beyond the error-correction bound. *J. Complexity*, *13*, 1, 180–193.

XU, L. AND BRUCK, J. 1999. X-code: MDS array codes with optimal encoding. *IEEE Trans. Inf. Theory 45*, 1.