LH*$_{RS}$ – A Highly-Available Scalable Distributed Data Structure

WITOLD LITWIN
U. Paris Dauphine
and
Rim Moussa
U. Paris Dauphine
and
Thomas J.E. Schwarz, S.J.,
Santa Clara University

LH*$_{RS}$ is a high-availability scalable distributed data structure (SDDS). An LH*$_{RS}$ file is hash partitioned over the distributed RAM of a multicomputer, e.g., a network of PCs, and supports the unavailability of any of its $k \geq 1$ server nodes. The value of k transparently grows with the file to offset the reliability decline. Only the number of the storage nodes potentially limits the file growth. The high-availability management uses a novel parity calculus that we have developed, based on the Reed-Salomon erasure correcting coding. The resulting parity storage overhead is about the minimal ever possible. The parity encoding and decoding are faster than for any other candidate coding we are aware of. We present our scheme and its performance analysis, including experiments with a prototype implementation on Wintel PCs. The capabilities of LH*$_{RS}$ offer new perspectives to data intensive applications, including the emerging ones of grids and of P2P computing.

Categories and Subject Descriptors: E.1 *Distributed data structures*, D.4.3 *Distributed file systems*, D.4.5 Reliability: *Fault-tolerance*, H.2.2 Physical Design : *Access methods, Recovery and restart*

General Terms: Scalable Distributed Data Structure, Linear Hashing, High-Availability, Physical Database Design, P2P, Grid Computing

---

Motto : Here is Edward Bear, coming downstairs now, bump, bump, bump, on the back of his head, behind Christopher Robin. It is, as far as he knows, the only way of coming downstairs, but sometimes he feels that there really is another way, if only he could stop bumping for a moment and think of it. And then he feels that perhaps there isn't. Winnie-the-Pooh. By A. A. Milne, with decorations by E. H. Shepard. Methuen & Co, London (publ.)

1 INTRODUCTION

Shared–nothing configurations of computers connected by a high-speed link, often called *multicomputers*, allow for high aggregate performance. These systems gain in popularity with the emergence of grid computing and P2P applications. They need new data structures that scale with the number of components [CACM97]. The concept of a Scalable Distributed Data Structures (SDDS) aims at this goal [LNS93]. An SDDS file transparently scales over multiple nodes, called the SDDS *servers*. As the file grows, so does the number of servers on which it resides. The SDDS addressing scheme has no centralized components. The speed of operations is then possibly independent of the file size. Many SDDS schemes are now known. They provide for hash, range or m-d

Authors' addresses: Witold Litwin, Rim Moussa,Université Paris 9 (Dauphine), Pl. du Mal. de Lattre, Paris 75016, France, Witold.Litwin@dauphine.fr, Rim.Moussa@dauphine.fr; Thomas J.E. Schwarz, S.J., Department of Computer Engineering, Santa Clara University, Santa Clara, mailto:tjschwarz@scu.edu.

partitioned files of records identified with a primary or with multiple keys. See [SDDS] for a partial list of references. A prototype system, SDDS 2000, for Wintel PCs, is freely available for a non-commercial purpose [CERIA].

Among best-known SDDS schemes is the LH* scheme [LNS93, LNS96, KLR96, BVW96, B99a, K98v3, R98]. It creates scalable distributed hash partitioned files. Each server stores the records in a bucket. The buckets split when the file scales up. The splits follow the *linear hashing* (LH) principles [L80a, L80b]. Buckets are usually stored in distributed RAM. Only the maximum number of nodes of the multicomputer limits the file size. A search or an insert of a record in an LH* file can be hundreds times faster than a disk access [BDNL00, B02].

An LH* server may become unavailable (failed), which makes it impossible to access its data. The likelihood of a server unavailability increases with the scaling file. Similarly, the likelihood of k unavailable servers for any fixed k increases with file size. Data loss or inaccessibility can be very costly [CRP06]. The well-known crash of EBay in June 1999 resulted in a loss of $4B of market value and of $25M in operations. The failure of a financial database may easily cost $10K-$27K per minute, [B99].

The information-theoretical minimum storage overhead for k-availability of m data servers is k/m [H&al94]. It requires the encoding of k parity symbols (records, buckets…), per m data symbols (records, buckets…). Decoding k unavailable symbols requires access to m available symbols among $m + k$. Large values for m seem impractical. One approach to reasonably limit m is to partition a data file into groups with independent parity calculus, of m nodes (buckets) at most per group.

For a small file using a few servers, a failure of more than one node is unlikely. Thus, $k = 1$ availability should typically suffice. The parity overhead is then the smallest, $1/m$, and the parity operations the fastest, using only XORing, as in RAID-5. The probability that a server becomes unavailable increases however with the size of the file. We need availability levels of $k > 1$ despite the increased storage overhead. Any static choice for k becomes eventually too small. The probability that k servers become unavailable increases necessarily. The file *reliability*, which is the probability that all the data are available for the application, declines necessarily as well. To offset this decline, we need the *scalable availability*, making k dynamically growing with the file [LMR98].

Below, we present an efficient scalable availability scheme we called LH*$_{RS}$. It generalizes the LH* scheme, structuring the scaling data file into groups of m data buckets, as we indicated above. The parity calculus uses a novel variant of Reed-Solomon (RS) erasure correcting coding/decoding we have designed. To our best

knowledge, it offers the fastest encoding for our needs. The storage overhead remains in particular in practice about optimal, between k/m and $(k+1)/m$.

We recall that RS codes use a parity matrix in a Galois Field (*GF*), typically in a *GF* (2^f). The *GF* (2^{16}) turned out to be the most efficient for us at present. The addition in a *GF* is fast, amounting to XORing. The multiplication is slower, regardless of the algorithm used [MS97]. Our parity calculus for $k = 1$ applies the XORing only, optimizing the most common case. We resort to *GF* multiplication only for $k > 1$. Our multiplication uses log and antilog tables. One novelty is that the XORing only encoding remains however then for the first parity symbol (record, bucket…), and for the decoding of a single unavailability of a data bucket. Another novelty is an additional acceleration of the encoding, by needing only XOR operations for the first bucket in each bucket group. Finally, we innovate by using a logarithmic parity matrix, as we will explain, accelerating the parity calculus even more. Besides, the high-availability management does not affect the speed of searches and scans. These operations perform as in an LH* file with the same data records and bucket size.

The study of LH*$_{RS}$ we present here stems from our initial proposals in [LS00]. The analysis reported below has perfected the scheme. This includes various improvements to the parity calculus, such as the use of *GF*(2^{16}) instead *GF*(2^8), as well as more extensive use of XORing and of the logarithmic parity matrices. We will show other aspects of the evolution in what follows. We have also completed the study of various operational aspects of the scheme, especially of messaging, crucial for the performance.

LH*$_{RS}$ is the only high-availability SDDS scheme operational to the extend we present, demonstrated by a prototype for Wintel PCs, [LMS04]. It is not however the only one known. There were proposals to use the mirroring for to achieve 1-availability [LN96, BV98, VBW98]. Two schemes using only XORing provide 1-availability [L&al97, LR01, L97]. Another XORing-only scheme LH*$_{SA}$ was the first to offer the scalable availability [LMRS99]. Its encoding speed can be faster than for LH*$_{RS}$. The price is sometimes greater storage overhead. We compare various schemes in the related work section.

We first describe the general structure of an LH*$_{RS}$ file and the addressing rules. Next, we discuss the mathematics of the parity calculus and its use for the LH*$_{RS}$ encoding and decoding. We then present the basic LH*$_{RS}$ file manipulations. We follow with a theoretical and experimental performance analysis of the prototype. Our measurements justify various design choices for the basic scheme and confirm its promising efficiency. In one of our experiments, about 1.5 sec sufficed to recover 100 000 records in three

unavailable data buckets, with more than 10MB of data. We investigate also variants with different trade-offs, including a different choice of an erasure correcting code. Finally we discuss the related work, the conclusions and the directions for future work. The capabilities of LH*$_{RS}$ appear to open new perspectives for data intensive applications, including the emerging applications of grids and of P2P computing.

Section 2 describes the LH*$_{RS}$ file structure. Section 3 presents the parity encoding. Section 4 discusses the data decoding. We explain the LH*$_{RS}$ file manipulations in Section 5. Section 6 deals with the performance analysis. In Section 7 we investigates variants to the scheme. Section 8 discusses the related work. Section 9 concludes the study and proposes directions for the future work. Appendix A shows our parity matrices for $GF(2^{16})$ and $GF(2^{8})$. Appendix B sums up our terminology.

2 THE LH*$_{RS}$ FILE STRUCTURE

LH*$_{RS}$ provides high availability to the LH* scheme [LNS93, LNS96, LMRS99]. LH* itself is the scalable distributed generalization of Linear Hashing (LH) [L80a, L80b]. An LH*$_{RS}$ file contains *data* records and *parity* records. Data records contain the application data. The application interacts with the data records as in an LH* file. Parity records provide high availability and are invisible to the application.

We store the data and parity records at the *server* nodes of the LH*$_{RS}$ file. The application does not access the servers directly, but uses the services of the LH*$_{RS}$ *client* component. A client usually resides at the application node. It acts as a middleware between the application and the servers.

An LH*$_{RS}$ operation is in *normal* mode as long as it does not access an unavailable bucket. If it does, the operation enters *degraded* mode. In what follows, we assume normal mode unless otherwise stated. We first present the storage and addressing of the data records. We introduce the parity management afterwards.

2.1 Data Records

2.1.1 Storage

An LH*$_{RS}$ file stores data records as if they constituted an LH*$_{LH}$ file, a variant of LH* described in [KLR96]. The details and correctness proofs of the algorithms presented below are in [KLR96] and [LNS96]. A data record consists of a (primary) *key* field, that identifies the record, and a *non-key* field, Figure 1. The application provides the data for both fields. We write *c* for the key. The records are stored in *data buckets*, numbered 0, 1, 2… and located each at a different *server* node of the multicomputer. The location of bucket *a*, i.e., the actual address *A* of the node supporting it, results from a mapping $a \rightarrow A$, e.g., through static or dynamic allocation tables at the clients and servers

[LNSb96]. A data bucket has the capacity to store $b>>1$ data records. Additional records become *overflow* records. Each bucket reports an overflow to the *coordinator* component of LH*. Typically, the coordinator resides at node of bucket 0.

Initially, an LH*$_{RS}$ file typically stores its data records only in bucket 0 at some server A_0. It also contains at least one parity bucket, at a different server, as we discuss later. The file adjusts to growth by dynamically increasing (or decreasing) the number of buckets. New buckets are created by splits. Inserts that overflow their buckets trigger these. Inversely, deletion causing an underflow may trigger bucket merges. Each merge undoes the last split, freeing the last bucket. The coordinator manages both splits and merges.

We show now how LH*$_{RS}$ data buckets split in normal mode. In Section 5, we treat the processing of the parity records during the splits, as well the degraded mode. We postpone the description of the merge operation to that section too.

As in LH file, the LH*$_{RS}$ buckets split in fixed order: 0; 0,1; 0,1,2,3; …; 0…n,…2^i-1; 0… The coordinator maintains the data (i, n) forming the *file state*. The variable i determines the hash function used to address the data records, as we discuss in Section 2.1.2 below. We call i the LH* *file level*. The variable n points to the bucket to split; we call it the *split* pointer. Initially, (i, n) = (0,0).

To trigger a split, the coordinator sends a *split message* to bucket n. It also dynamically appends a new bucket to the file with the address $n + 2^i$. The address of each key c in bucket n is then recalculated using a hash function h_{i+1}: $c \rightarrow c \bmod 2^{i+1}$. We call the functions h_i *linear hash functions* (LH-function). For any record in bucket n, either $h_{i+1}(c) = n$ or $h_{i+1}(c) = n + 2^i$. Accordingly, any record either remains in bucket n or migrates to the new bucket $n + 2^i$. Assuming as usual that key values are randomly distributed, both events are equally likely.

After the split, the coordinator updates n as follows. If $n < 2^i - 1$, it increments n to $n + 1$. Otherwise it sets it to $n = 0$ and increments i to $i + 1$.

Internally, each LH*$_{RS}$ data bucket a is organized as an LH file. We call the internal buckets *pages*. There is also an internal file state, called the *bucket state,* ($\tilde{\imath}$, $\tilde{n}$). We perform the LH*$_{RS}$ split, (and the LH*$_{LH}$ split in general [KLR96]), by moving all odd pages to the new bucket a'. This is faster than the basic technique of recalculating the address for every record in bucket a. We rename the odd pages in the new bucket a' to 0, 1… by dropping the least significant bit. Likewise, we rename the even pages in bucket a. In other words, page number p becomes page number $p/2$ in the new buckets.

We also decrement the level l of the pages by one. The internal file state in the new bucket a' becomes $(\tilde{\imath} - 1, \lfloor \tilde{n}/2 \rfloor)$. We change the file state of the splitting bucket to $(\tilde{\imath}, \tilde{n}) := (\tilde{\imath}, 0)$ if $\tilde{n} = 2^{\tilde{\imath}} - 1$, and to $(\tilde{\imath}, \tilde{n}) := (\tilde{\imath} - 1, \lfloor \tilde{n}+1 \rfloor/2)$ otherwise. See [KLR96] for details. We deal with page merges accordingly.

2.1.2 Addressing

The LH* storage rules guarantee that for any given file state (i, n), the following *LH addressing algorithm* [L80a] determines the bucket a for a data record with key c uniquely:

(A1) $a := h_i(c)$; **if** $a < n$ **then** $a := h_{i+1}(c)$.

Algorithm (A1) determines the *correct* (primary) address for key c. Its calculus depends on the file state at the coordinator. The general principles of SDDS mandate avoiding hot spots. Therefore, LH*$_{RS}$ clients do not access the coordinator to obtain the file state. Each client uses instead (A1) on its private copy of the file state. We call it *client image*, and write as *(i',n')*. Initially, for a new client or file, $(i', n') = (0,0)$, as the initial file state. In general, the client image differs from the file state, as the coordinator does not notify the clients of any bucket splits and merges. The strategy reflects the basic SDDS design, avoiding excessive messaging after a split, problems with unavailable clients, etc. [LNS96]. As the result, any split or merge causes all client images to be out of date.

Using (A1) on the client image can lead to an *incorrect* address. The client then sends the request to an incorrect bucket. The receiving bucket forwards the request that normally reaches the correct server in one or at most two additional hops (as we will show more below). The correct LH*$_{RS}$ server sends an *image adjustment message (IAM)*. The IAM adjusts the image so that the same addressing error cannot occur again. It does not guarantee that the image becomes equal to the file state. In general, different clients may have different images.

The binding between bucket numbers and node addresses is done locally as well. Buckets can change their location. For instance, a bucket can merge with its "father" bucket, and then again split off, but on another (spare) server. A bucket located on a failed node is also reconstructed on a spare. We call a bucket, and query to it, *displaced* if the bucket is located at a different server than the client knows. We use IAM message to update these data whenever the client deals with a displaced bucket. It will appear that the presence of displaced bucket may lead to one or two additional hops.

Most LH*$_{RS}$ file operations are key-based. The exception is the scan operation, which

returns all records satisfying a certain condition. We treat scans in Section 5.5. The key-based operations are *insert*, *delete*, *update*, and (key) *search*. To perform such an operation, the client uses (A1) to obtain a bucket address a_1 and sends the operation to this bucket. Most of the time [LNS96], the correct bucket number $\tilde{a}$, is identical to a_1. However, if the file grew and now contains more buckets, then $a_1 < \tilde{a}$ is possible. If the file shrank, then $a_1 > a$ is possible as well. In both cases, the client image differed from the file image.

To be able to resolve the incorrectly addressed operations, every bucket stores the j value of h_j last used to split or to create the bucket. We call j the bucket *level* and we have $j = i$ or $j = i + 1$. A server that receives a message intended for bucket $\tilde{a}$, first tests whether it really has bucket $\tilde{a}$. If not, it forwards the displaced query to the coordinator. The coordinator attempts to resolve the addressing using the file state. We treat this case later in this section.

Otherwise, bucket $\tilde{a}$ starts the LH* *forwarding* algorithm (A2):

(A2) $a' := h_j(c)$;
if $a' = \tilde{a}$ **then** accept c ;
else $a'' := h_{j-1}(c)$;
if $a'' > \tilde{a}$ **and** $a'' < a'$ **then** $a' := a''$;
send c **to bucket** a' ;If the address a' provided by (A2) is not $\tilde{a}$, then the client image was incorrect. Then, bucket $\tilde{a}$ forwards the query to bucket $a_2 = a'$, $a_2 > a_1$. If the query is not displaced, bucket a_2 becomes the new intended bucket for the query, i.e., $\tilde{a} := a_2$. It acts accordingly, i.e., executes (A2). This may result in the query forwarded to yet another bucket $a_3 := a'$, with a' recalculated at bucket a_2 and $a_3 > a_2$. Bucket a_3 becomes the next intended bucket, i.e., $\tilde{a} := a_3$. If the query is not displaced, bucket a_3 acts accordingly in turn, i.e., it executes (A2). A basic property of LH* scheme is then that (A2) must yield $a' = a_3$, i.e., bucket a_3 must be the correct bucket a. In other words, the scheme forwards a key-based operation at most twice.

The correct bucket a performs the operation. In addition, if it received the operation through forwarding i.e. $a \neq a_1$, it sends an IAM to the client. The IAM contains the level j of bucket a_1 (as well as the locations known to bucket a, which are in fact all those of its preceding buckets). The client uses the *IAM* to update its image as follows, according to the LH* *Image Adjustment* algorithm:

(A3) **if** $j > i'$ **then** $i' := j - 1$, $n' := a + 1$;
if $n' \geq 2^{i'}$ **then** $n' = 0$, $i' := i' + 1$;

(A3) guarantees that the client cannot repeat the same error, although the client image typically still differs from the file image. The client also updates its location data.

We process the query within the correct bucket as a normal LH query. We locate first the page that contains the key *c*. However, if the bucket has level $j > 0$, we do not apply the addressing algorithm (A1) directly to *c*, but rather to *c* shifted by *j* bits to the right. In other words, we apply (A1) to $\lfloor c/2^j \rfloor$. The modification corresponds to the algorithm for splitting buckets described in the previous section, [KLR96].

If a server forwards a displaced query to the coordinator, the latter calculates the correct address *a*. It does so according to (A1) and the file state. It sends the actual location of the displaced bucket to the query originator and to the server. These update their locations accordingly.

2.2 Parity Records

The LH*$_{RS}$ parity records protect an application against the unavailability of servers with its stored data. The LH*$_{RS}$ file tolerates up to $k \geq 1$ unavailable server in a manner transparent to the application. As usual, we call this property *k-availability*. The (actual) *availability level k* depends on an LH*$_{RS}$ file specific parameter, the *intended availability level* $K \geq 1$. We adjust *K* dynamically with the size of the file, (Section 2.2.3). Depending on the file state, the actual availability level *k* is either *K* or *K*–1.

2.2.1 Record Grouping

LH*$_{RS}$ parity records constitute a specific structure, invisible to the application, and separate from the LH* structure of the data records. The structure consists of *bucket groups* and *record groups*. We collect all buckets *a* with $\lfloor a / m \rfloor = g$ in a bucket group *g*, $g = 0, 1 \ldots$ Here, $m > 0$ is a file parameter that is a power of 2. In practice, we only consider $m \leq 128$, as we do not envision currently any applications where a larger value might be practical. A bucket group consists thus of *m* consecutively numbered buckets, except perhaps for the last bucket group with less than *m* members.

Data records in a bucket group form *record groups*. Each record group (*g,r*) is identified by a unique *rank r*, and the bucket group number *g*. At most one record in a bucket has any given rank *r*. The record gets its rank, when an insert or split operation places it into the bucket. Essentially, records arriving at a bucket are given successive ranks, i.e. $r = 1, 2\ldots$ However, we can reuse ranks of deleted records. A record group contains up to *m* data records, each at a different bucket in the bucket group. A record that moves with a split obtains a new rank in the new bucket. For example, the first record arriving at a bucket *a*, $a = 0, 1 \ldots m–1$, because of an insert or a split, obtains rank 1. It thus joins the record group (0,1). If the record is not deleted, before the next record arrives at that bucket, that one joins group (0,2), etc.

Each record group has $k \geq 1$ parity records $p_1 \ldots p_k$ in addition to the data records. The parity records are stored at different *parity buckets* $P_1, \ldots P_k$. The *local availability level k* depends on the intended availability level *K*, the bucket group number *g*, and the file state. Using the *parity calculus* presented in Section 3 and given any $s \leq k$ parity records and any *m-s* data records, we can recover the remaining *l* data records in the group. The availability level of the file is the minimum of the local availability levels.

2.2.2 Record Structure

Figure 1 (ii) shows the structure of a parity record for group (*g*,*r*). The first field of every record p_i of the group contains the rank *r* as the record key in P_i. The next field is the (*record*) *group structure* field $C = c_0, c_1, \ldots c_{m-1}$. If there is a data record in the i^{th} bucket of the group, then c_i is the key of that data record. Otherwise, c_i is zero.

The final field is the *parity* field *B*. The contents of *B* are the *parity symbols.* We calculate them from the non-key data in the data records in the record group (Section 3) in a process called *encoding*. Inversely, given *s* parity fields and *m-s* non-key data fields from records in the group, we can recover the remaining non-key data fields using the *decoding* process from Section 3. We can recover the keys of the *s* lacking data records from any parity record.

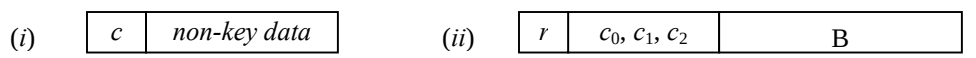

Figure 1: LH*$_{RS}$ record structure: (*i*) data record, (*ii*) parity record.

In [LS00], we used a variable length list of keys for existing data structures. The fixed structure presented here and introduced since in [Lj00], proved however more efficient. It typically needs slightly less storage. In addition, its position indicates the bucket at which data records are located. During record recovery, we can directly access the bucket instead of using the LH* addressing algorithm. We avoid possible forwarding messages.

2.2.3 Scalable Availability

Storing and maintaining parity creates storage overhead increasing with *k*. For a file with only 1 data bucket, the overhead is *k*. For a larger file, the overhead is at least equal to *k/m.* In addition, we have the run-time overhead to update all *k* parity records of a group whenever the application inserts, updates or deletes a data record. A file with few buckets is less likely to suffer from multiple unavailable buckets. However, as the size of the file increases, multiple failures become more probable. For any given *k*, the probability of catastrophic loss, i.e. loss of more than *k* buckets in a single group and the resulting inability to access all records, increases with the file size [H&a94]. In response,

the LH*$_{RS}$ scheme provides the *scalable availability*, [LMR98]. When the growing file reaches certain sizes, the file starts to incrementally increase every local *k*-availability to *k*+1-availability. We illustrate the principle in Figure 2 that we will discuss more in depth soon.

Specifically, we maintain the file parameter termed *intended availability level K*. If we create a new bucket that is the first in the group, then this group gets $k = K$ parity buckets. Every data record in this group has then $k = K$ parity records. Initially, $K = 1$, and any group has one parity bucket. We basically increment K when (*i*) the split pointer returns to bucket 0, and (*ii*) the total number of buckets reaches some predetermined level. Then, any existing bucket group has $k = K – 1$ parity buckets (as we will see by induction). Every new group gets then K parity buckets. In addition, whenever we split the first bucket in a group, we equip this group with an additional parity bucket. As the split pointer moves through the data buckets of the file, we create all new groups with K parity buckets, and add an additional parity bucket to all old groups. Thus, by the time the split pointer reaches bucket 0 again, all groups have local availability level $k = K$.

In Figure 2, a bucket group has the size of $m = 4$. Data buckets are white and parity buckets are grey. In Figure 2a, we create the file with one data bucket and one parity bucket, i.e., with $K = 1$. When the file size increases, we split the first bucket, but only maintain one parity bucket, Figure 2b,c. When the m+1st bucket is created, the new bucket group receives also receives a parity bucket. Thus, the file is 1-available. Each new bucket group has this availability level until the number N of data buckets in the file reaches some N_1. In our example, $N_1 = 16$, Figure 2e. More generally, $N_1 = 2^l$ with some large $l >> 1$ in practice. That condition implies $n = 0$. The next bucket to split is bucket 0.

At this point of the file scale up, K increases by one. From now on, starting with the split of bucket 0, each split creates two parity records per record group, Figure 2f. Both the existing group and the one started by the split have 2 parity buckets. The process continues until some size N_2, also a power of 2 and, necessarily this time, a multiple of m. On the way, starting from the file size $N = 2N_1$, all the bucket groups are 2-available, Figure 2g. When the file grows to include N_2 buckets, K increases by 1 again, to $K = 3$ this time. The next split adds a third parity bucket to the group of bucket 0 and initializes a new group starting with bucket N_2 and carrying three parity buckets. The next series of splits provides all groups with $K = 3$ parity records. When the file size reaches N_3, K is again incremented, etc.

Basically for our scheme, and in Figure 2, the successive values N_i are predefined as

$N_{i+1} = 2^i N_1$. We call this strategy *uncontrolled availability* and justify it in [LMR98]. Alternatively, a *controlled availability* strategy implies that the coordinator calculates dynamically the values of the N_i. The decision may be based on the probability of k unavailable buckets in a bucket group, whenever the split pointer n comes back to 0.

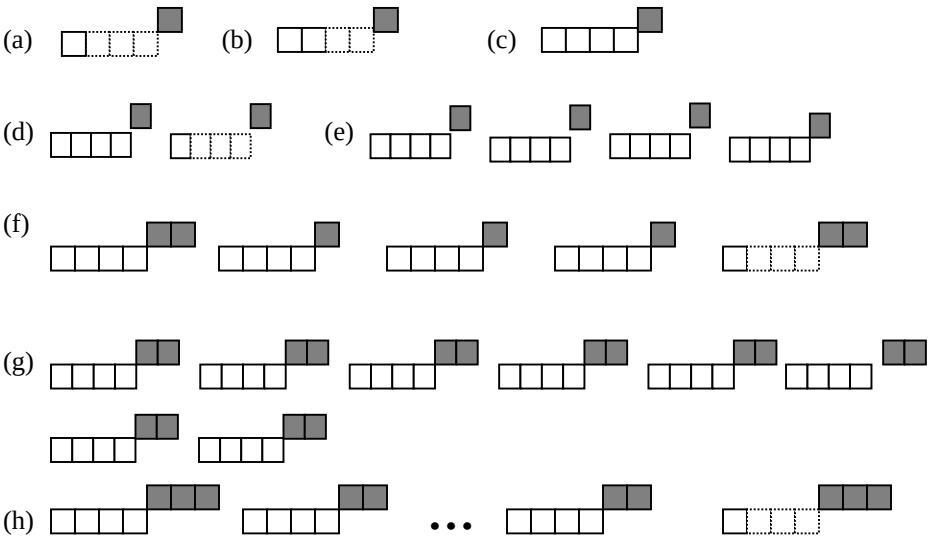

Figure 2: Scalable Availability of LH*$_{RS}$ File.

The *global file availability level* K_{file} is the maximum k so that we can recover any k buckets failing simultaneously. Obviously, K_{file} is equal to the minimum k for all groups in the file. We thus have $K_{file} = K$ or $K_{file} = K–1$. K_{file} starts at 1 and increases to $K_{file} = 2$, when N reaches $2N_1$. In general, K increases to i after N reaches N_i and K_{file} reaches i when N reaches $2N_i$. The growing LH*$_{RS}$ file is thus progressively able to recover from larger and larger numbers of unavailable buckets, as these events become increasingly likely, necessarily.

One consequence of the scheme is the possible presence of *transitional* bucket groups where not all the data buckets are split yet. The split pointer n points there somewhere between the 2nd and the last bucket of the group. The first bucket group in Figure 2f is transitional, as well as in Figure 2h, both with $n = 2$. In such a group, the newly added parity bucket only encodes the contents of the data buckets that have already split. LH*$_{RS}$ recovery cannot use this additional parity bucket in conjunction with data buckets that have not yet split. As the result, the availability level of any transitional group is $K – 1$. It becomes K when the last bucket splits, (hence the group ceases to be transitional).

3 PARITY ENCODING

We now explain our parity encoding, that is, the calculation of the B field in a parity

record. Section 4 deals with the decoding for the reconstruction of an unavailable record. The parity encoding in general is based on Erasure Correcting Codes (ECC). We have designed a generalization of a Reed-Solomon (RS) code. RS codes are popular, [MS97], [P97], being sometimes indirectly referred to also as *information dispersal* codes, [R89]. Other codes are a possibility, we discuss the trade-offs in Section 7.5. Our parity calculations are operations in a Galois Field (*GF*) as detailed in Section 3.1. We use a *parity matrix* (Section 3.2), which is a submatrix of a *generator* matrix (Section 4.1).

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	26	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	108	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

Table 1: Logarithms for *GF*(256).

3.1 Galois Field

Our *GF* has 2^f elements ; $f = 1,2\ldots$, called *symbols*. Whenever the *size* 2^f of a *GF* matters, we note 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, consisting of f zero-bits. Another is the *one* symbol, written as 1, with f-1 bits 0 followed by bit 1. Symbols can be added (+), multiplied (·), 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 '·' symbol.

Initially, we elaborated the LH*$_{RS}$ scheme for $f = 4$, [LS00]. First experiments showed that $f = 8$ was more efficient. The reason was the (8-bit) byte and word oriented structure of current computers [Lj00]. Later, the choice of $f = 16$ proved even more practical. It became our final choice, Section 6.3. For didactic purposes, we discuss our parity calculus nevertheless for $f = 8$, i.e., for $GF(2^8) = GF(256)$. The reason is the sizes of the tables and matrices involved. We note this GF as F. The symbols of F are all the byte values. F has thus 256 symbols which are 0,1…255 in decimal notation, or 0,1...ff in hexadecimal notation. We use the latter in Table 1 and often in our examples.

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

$a + b = a - b = b - a = a \oplus b = a$ XOR b.

The XOR operation is widely available, e.g., as the ^ operator in C and Java, i.e., a XOR $b = a \wedge b$. The multiplication and division are more complex operations. There are different methods for their calculus. We use a variant of the *log/antilog* table calculus [LS00], [MS97].

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

Figure 3: Galois Field Multiplication Algorithm.

The calculus exploits the existence in every GF of the *primitive* elements. 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 1 tabulates the non-zero $GF(2^8)$ elements and their logarithms for $\alpha = 2$. Likewise, $\xi = \alpha^i$ is then the *antilogarithm* of i that we write as $\xi = \text{antilog}(i)$.

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 $i' = i \mod 2^f - 1$. Using the logarithms and the antilogarithms, we can calculate multiplication and division through the following formulae. 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 formulae is the usual one of integers:

$\xi \cdot \psi = \text{antilog}(\log(\xi) + \log(\psi) \mod (2^f - 1))$,

$\xi / \psi = \text{antilog}(\log(\xi) - \log(\psi) + 2^f - 1 \mod (2^f - 1))$.

To implement these formulae, we store symbols as char type (byte long) for $GF(2^8)$ and as short integers (2-byte 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 1 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 mod (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 are rare and the savings seem too minute to justify the additional storage (128KB for our final choice of f = 16). Figure 3 shows our final multiplication algorithm. Figure 4 shows the algorithm generating our two arrays. We call them respectively `log` and `antilog` arrays. The following example illustrates their use.

Example 1

We use the result of the following $GF(2^8)$ calculation later in Example 2:

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

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 1). We add these up as integers to obtain 257. This value is not in Table 1, 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 here 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 1 it contains 51 (in hex), which is the final result.

3.2 Parity Matrix

3.2.1 Parity Calculus

We recall that a parity record contains the keys of the data records and the parity data of the non-key fields of the data records in a record group, Figure 1. We encode the parity data from the non-key data as follows.

We number the data records in the record group 0, 1,… *m*-1. We represent the non-key field of the data record *j* as a sequence $a_{0,j}, a_{1,j}, a_{2,j} \ldots$ of symbols. We give to all the records in the group the same length *l* by at least formally padding with zero symbols if necessary. If the record group does not contain *m* records, then we (conceptually) replace the missing records with dummy records consisting of *l* zeroes.

```
#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];
}
```

Figure 4: Calculus of tables `log` and `antilog` for $GF(2^f)$.

We consider all the data records then in the group as the columns of an *l* by *m* matrix $\mathbf{A} = (a_{i,j})$. We also number the parity records in the record group 0,1…*k*. We write $b_{0,j}, b_{1,j}, b_{2,j}, \ldots$ for the B-field symbols of the j^{th} parity record. We arrange the parity records also in a matrix with *l* rows and *k* columns $\mathbf{B} = (b_{i,j})$. Finally, we consider the *parity* matrix $\mathbf{P} = (p_{\lambda,\mu})$ that is a matrix of symbols *p* forming *m* rows and *k* columns. We show the construction of **P** in next sections. Its key property is the linear

relationship between the non-key fields of data records in the group and the non-key fields of the parity records:

$$\mathbf{A} \cdot \mathbf{P} = \mathbf{B}.$$

More in depth, each j^{th} row of **A** is a vector $\mathbf{a}^j = \left(a_{j,0}, a_{j,1}, \ldots, a_{j,m-1}\right)$ of the symbols in all the successive data records with the same offset *j*. Likewise, every j^{th} line $\mathbf{b}^j$ of **B** contains the parity symbols with the same offset *j* in the successive parity records of the record group. The above relationship means that:

$$\mathbf{b}^j = \mathbf{a}^j \mathbf{P}.$$

Each parity symbol is thus the sum of *m* products of data symbols with the same offset times *m* coefficients of a column of the parity matrix:

$$b_{j,\lambda} = \sum_{\nu=0}^{m-1} a_{j,\nu} \cdot p_{\nu,\lambda} \ . \tag{3.1}$$

The LH*$_{RS}$ parity calculus does not use **P** directly. Instead, we use the *logarithmic parity matrix* **Q** with coefficients $q_{i,j} = \log_\alpha(p_{i,j})$. The implementation of equation (3.1) gets the form:

$$b_{\iota,\lambda} = \oplus_{\nu=0}^{m-1} \text{antilog}(q_{\nu,\lambda} + \log(a_{\iota,\nu})) \tag{3.2}$$

Here, $\oplus$ designates XOR and the antilog designates the calculus using our `antilog` table, which avoids the mod (2^f–1) computation. Using (3.2) and **Q** instead of (3.1) and **P** speeds up the encoding, by avoiding half of the accesses to the `log` table. The overall speed-up of the encoding is however more moderate than one could perhaps expect from these figures (Section 6.3.1). While using **Q** that is our actual approach, we continue to present the parity calculus in terms of **P** for ease of presentation.

3.2.2 Generic Parity Matrices

We have designed for LH*$_{RS}$ several algorithms for generating parity matrices. We presented the first one in [LS00] for 4-bit symbols of $GF(2^4)$. When implemented, operations turned out to be slower they could be on the byte-oriented structure of modern computers [Lj00]. We turned to byte sized symbols of $GF(2^8)$ that proved faster, and to 2-byte symbols of $GF(2^{16})$ that proved even more effective. We have reported early results in [M03]. We show further outcomes below.

We upgraded our parity matrices with respect to [LS00] (and any other proposal we know about in the literature) so that the first column and the first row now only contain coefficients 1. The column of ones allows us to calculate the first parity records of the bucket group using the XOR only, as for the "traditional" RAID-like parity calculus. Our

prior parity matrices required *GF* multiplications for this column, slower than XOR alone as we already discussed. Next, if one data bucket in a group has failed and the first parity bucket is available, then we can decode the unavailable records using XOR only. Before, we also needed the *GF* multiplications. The row of ones allows us to use XOR calculations for the encoding of each first record of a record group. This also contributes to the overall speed up as well, with respect to any proposal requiring the multiplications, including our own earlier ones. Our final change was the use of **Q** instead of the original **P** (Section 3.2.1). The experiments confirmed the interest of all these changes (Section 6.3.1).

LH*$_{RS}$ files may differ by their group size m and availability level k. Smaller m speed up the recovery time, but increase the storage overhead, and vice versa. The parity matrix **P** for a bucket group needs m rows and k columns, $k = K$ or $k = K - 1$. Different files in a system may need in this way different matrices **P**. We show in Section 4.2 that the choice of $GF(2^f)$ limits the possibilities for any **P** to $m + k \leq 2^f + 1$. Except for this constraint, m and k can be chosen quite arbitrarily. We also prove that for any parity matrix **P'** with dimensions m' and k', every $m < m'$ by $k < k'$ top left corner of **P'** is also a parity matrix. These properties govern our use of the parity matrices for different files. Namely, we use a *generic* parity matrix **P'** and its logarithmic parity matrix **Q'** in an LH*$_{RS}$ file system. The m' and k' dimensions of **P'** and **Q'** should be big enough for any system application. Any actual **P** and **Q** we use are then the $m \leq m'$ by $k \leq k'$ top left corners of **P'** and of **Q'**. Their columns are derived dynamically when needed.

Section 4.2 below shows the construction of our **P'** for $GF(2^8)$, (within the generator matrix containing it). We have to respect the condition that $m' + k' \leq 257$. Because of LH*$_{RS}$ specifically, m' has to be a power of two. Our choice for m' was therefore $m' = 128$, to maximize the bound on the group size while allowing $k' > 1$. Hence, $k' = 129$. Figure 16 displays the 20 leftmost columns of **Q'**. Figure 17 displays these columns of **P'**. The selection suffices for 20-available files. We are not aware of any application that needs higer level of availability.

As we said, we finally applied $GF(2^{16})$. **P'** may then reach $m' = 32K$ by $k' = 32K + 1$. This allows LH*$_{RS}$ files with more than 128 buckets per group. Ultimately, even a very large file could consist of a single group, if such an approach would ever prove useful.

Example 2

We continue to use $GF(2^8)$ and the conventions of Section 3.1. We now illustrate the encoding principles presented until now, by the determination of the parity data that

should be in k =3 parity records for the record group of size of m = 4 whose description follows. Figure 5 shows **P** and **Q.** These are the top left four rows and three columns of **P'** and **Q'** in Figure 17 and Figure 16.

$$\mathbf{P} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1a & 1c \\ 1 & 3b & 37 \\ 1 & ff & fd \end{pmatrix} \qquad \mathbf{Q} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 105 & 200 \\ 0 & 120 & 185 \\ 0 & 175 & 80 \end{pmatrix}$$

Figure 5: Matrices P and Q derived from P' and Q' for a 3-available record group of size m = 4.

We suppose the encoded data records to have the non-key fields as follows: "En arche en o logos …", "In the beginning was the word …", "Au commencement était le mot …", and "Am Anfang war das Wort…". Using ASCII coding, these strings translate to (hex) strings of our *GF* symbols: "45 6e 20 61 72 63 68 …", "49 6e 20 74 68 65 20 …", "41 75 20 63 6f 6d 6d …", and "41 6d 20 41 6e 66 61 …" To calculate the first parity symbols, we form the vector $\mathbf{a^0}$ = (45,49,41,41) and multiply it with **P.** The result $\mathbf{b^0}$ = $\mathbf{a^0 \cdot P}$ is (c, d2, d0). We calculate the 1st symbol of $\mathbf{b^0}$ simply as 45 + 49 + 41 + 41 = c. This is the conventional parity, as in a RAID. The calculation of the parity second symbol is in fact given in Example 1. We use formally the second column of **P** and the *GF* multiplication to obtain:

$$\begin{aligned} \begin{pmatrix} 45 & 49 & 41 & 41 \end{pmatrix} \bullet \begin{pmatrix} 1 & 1a & 3b & ff \end{pmatrix}^T &= 45 \cdot 1 + 49 \cdot 1a + 41 \cdot 3b + 41 \cdot ff \\ &= 45 + 4 + 5d + ce \\ &= d2. \end{aligned}$$

In our implementation, we use **Q** and the '*' multiplication between two *GF* elements (or matrices) when the right operand is a logarithm. This yields, according to Table 1:

$$\begin{aligned} &\begin{pmatrix} 45 & 49 & 41 & 41 \end{pmatrix} * \begin{pmatrix} 0 & 105 & 120 & 175 \end{pmatrix}^T \\ &= 45 * 0 + 49 * 105 + 41 * 120 + 41 * 175 \\ &= 45 + \text{antilog}(\log(49)+105) + \text{antilog}(\log(41)+120) + \text{antilog}(\log(41)+175) \\ &= 45 + \text{antilog}(152 + 105) + \text{antilog}(191 + 120) + \text{antilog}(191+175) \\ &= 45 + \text{antilog}(257) + \text{antilog}(311) + \text{antilog}(191+175) \\ &= 45 + 04 + 5d + ce \\ &= d2 \end{aligned}$$

Analogously, we get the last element in $\mathbf{b^0}$. The calculus iterates for $\mathbf{b^1}$ = (18,76,93), $\mathbf{b^2}$ = (0,e2,ff) …. As the result, the B-field in the 1st parity record, Figure 1, is encoded as B = "c 18 0…". Likewise, the second B-field is "d2 76 e2…" and the third one is finally "d0 93 ff…".

3.3 Parity Updating

The application updates an LH*$_{RS}$ file a data record at a time. An insert, update or delete of a record, modifies the parity records of a record group. We call this *parity updating*. It is the actual calculus for the parity encoding in the LH*$_{RS}$ files. We now introduce these principles.

We formally assimilate an insert and a deletion to specific cases of the update that is here our generic operation. Recall, we only operate in fact on the non-key fields. An insert changes the record from a zero string. A deletion does the opposite. We now consider an update in this way to the i^{th} data record in its group. Let matrix **A** of vectors **a** in Section 4.2 above be the symbols in the data records in the record group before the update. Let matrix **A'** contain the symbols after the update. The matrices only differ in the i^{th} column. Let **B** and **B'** be the matrices of vectors **b** with the resulting parity codes.

The codes should conform to the generic calculus rules in Section 4.2. We thus have **B = AP**, **B' = A'P**. The difference **B - B'** is Δ **= (A - A') P**. We have **A - A' =** $(0,...,0,\Delta_i,0,...,0)$ where Δ_i is the column with the differences between the same offset symbols in the former and new records. To calculate Δ = (**A - A'**)**P**, we only need the i^{th} row of **P.** Since **B' = B +** Δ, we calculate the new parity values by calculating Δ first and then XOR this to the current **B** value. In other words, with $\mathbf{P}_i$ being the i^{th} row of **P**:

(3.3) $$\mathbf{B'} = \mathbf{A'}\cdot\mathbf{G} = (\mathbf{A}+(\mathbf{A'}-\mathbf{A}))\,\mathbf{P} = \mathbf{A}\,\mathbf{P} + (\mathbf{A} - \mathbf{A'})\,\mathbf{P} = \mathbf{P} + \Delta_i\,\mathbf{P}_i.$$

In particular, if b_j is the old symbol, then we calculate the new symbol b'_j in record j as

(3.4) $$b'_j = b_j + \Delta_i\, p_{i,j},$$

where Δ_i is the difference between the new and the old symbol in the updated record, and $p_{i,j}$ is the coefficient of **P** located in the i^{th} row and j^{th} column.

The Δ-*record* is the string obtained as the XOR of the new and the old symbols with the same offset within the non-key field of the updated record. For an insert or a delete, the Δ-record is the non-key data. We implement the parity updating operation resulting from an update of a data record with key c and rank r as follows. The LH*$_{RS}$ data bucket computes the Δ-record and sends it, together with c and r, to all the parity buckets of the record group. Each bucket sets the B field value according to (3.3). It then either updates the existing parity record r or creates it. Likewise, the data record deletion updates the B field of the parity record r or removes all records r in each parity bucket. We discuss these operations more in depth in Section 5.6 and 5.7.

As we have seen, the l^{th} parity bucket in a bucket group only needs the column $\mathbf{p}^l$ of **P**

for the encoding. A parity bucket stores therefore basically only this column. Obviously, the first parity bucket does not have to store $\mathbf{p}^1$ if it is the column of ones as above.

Notice that our parity updating needs only one data record in the group, i.e., the updated one. This property is crucial to the efficiency of the encoding scheme. In particular, update speed is independent of *m*. Its theoretical basis is that our coding scheme is *systematic*. We elaborate more on it while discussing alternate codes in Section 7.5.1.

Example 3

We continue with the running example. We consider a file of four data buckets D0, D1, D2, and D3 forming the bucket group of size $m = 4$. We also consider three parity buckets P0, P1, P2 corresponding to the columns of matrices **P** and **Q** in Figure 5. We now insert one by one the records from Example 2. We assume they end up in successive buckets and form a record group. At the end, we also update the record in D1. Figure 6 shows vertically each non-key field of a data record in the group, and the evolution of the B-fields, also represented vertically. It thus illustrates also the matrices **A, A'** and **B,** and **B'** for each parity updating operation we perform.

Figure 6a shows the insert into D0 of the 1st record, with non-key data "En arche ...", i.e., (hex) "45 6e 20 61 72 63 68 65 20 65 ...". The Δ-record is identical to the record, being the difference between this string and the previous non-key data string, which is here the zero string. The first row of **P** consisting of ones, we calculate the content of each parity bucket by XORing the Δ-record to its previous content. As there were no parity records for our group yet, each B-field gets the Δ-record and we create all three records.

Figure 6b shows the evolution after the insert of "In principio …" into D1. The Δ-record is again identical to the data record. At P0, the existing parity record is XORed with the Δ-record. At P1, we multiply the Δ-record by '1a' and we XOR the result with the existing string. The '1a' is the **P**-coefficient located in Figure 5 in the second row (corresponding to D1) and the second column (corresponding to P1). We update the parity data in P2 similarly, except that we multiply by '1c'.

Figure 6c-d show the evolution after inserts of "Am Anfang war …" into D2 and "Dans le commencement …" into D3. Finally, Figure 6e shows the update of the record in D0 to "In the beginning was …". Here, the Δ-record is the XOR of "49 6e 20 74 68 65 20 62 65 67 …" and of "45 6e 20 61 72 63 68 65 20 65", yielding "c 0 0 15 1a 6 48 7 45 67 …". We send this Δ-record to the parity buckets. It comes from D0, so we only XOR

the Δ-record to the strings already there.

(a)

D0	D1	D2	D3	P0	P1	P2
45	0	0	0	45	45	45
6e	0	0	0	6e	6e	6e
20	0	0	0	20	20	20
61	0	0	0	61	61	61
72	0	0	0	72	72	72
63	0	0	0	63	63	63
68	0	0	0	68	68	68

(b)

D0	D1	D2	D3	P0	P1	P2
45	49	0	0	c	41	ea
6e	6e	0	0	0	4b	32
20	20	0	0	0	47	87
61	70	0	0	11	75	48
72	72	0	0	0	52	63
63	69	0	0	a	0	6b
68	6e	0	0	6	4d	34

(c)

D0	D1	D2	D3	P0	P1	P2
45	49	41	0	4d	1c	9c
6e	6e	6d	0	6d	c	93
20	20	20	0	20	74	29
61	70	41	0	50	28	3e
72	72	6e	0	6e	58	9b
63	69	66	0	6c	cf	36
68	6e	61	0	67	23	ec

(d)

D0	D1	D2	D3	P0	P1	P2
45	49	41	44	9	f6	fe
6e	6e	6d	61	c	54	09
20	20	20	6e	4e	40	c1
61	70	41	73	23	d8	28
72	72	6e	20	4e	ce	4d
63	69	66	6c	0	18	39
68	6e	61	65	2	a0	a5

(f)

D0	D1	D2	D3	P0	P1	P2
49	49	41	44	5	fa	f2
6e	6e	6d	61	c	54	9
20	20	20	6e	4e	40	c1
74	70	41	73	36	cd	3d
68	72	6e	20	54	d4	57
65	69	66	6c	6	1e	3f
20	6e	61	65	4a	e8	ed

Figure 6: Example of Parity Updating Calculus.

4 DATA DECODING

4.1 Using Generator Matrix

The decoding calculus uses the concept of a generator matrix. Let **I** be an m x m identity matrix and **P** a parity matrix. The *generator* matrix **G** for **P** is the concatenation **I**|**P**. We recall from Section 3.2.1 that we organize the data records in a matrix **A**. Let **U** denote the matrix **A**·**G**. **U** is the concatenation (**A**|**B**) of matrix **A** and matrix **B** from the previous section. We refer to each line $\mathbf{u} = (a_1, a_2, ..., a_m, a_{m+1}, ..., a_n)$ of **U** as a *code word*. The first m coordinates of **u** are the coordinates of the corresponding line vector **a** of **A**. We recall that these are the data symbols with the same offset in all the data records

in the record group. The remaining *k* coordinates of **u** are the newly generated parity codes. A column **u'** of **U** corresponds to an entire data or parity record.

A crucial property of **G** is that any *m* by *m* square submatrix **H** is invertible. (See Section 4.2 for the proof.) We use this property for reconstructing up to *k* unavailable data or parity records. Consider first that we wish to recover only data records. We form a matrix **H** from any *m* columns of **G** that do not correspond to the unavailable records. Let **S** be **A·H**. The columns of **S** are the *m* available data and parity records we picked in order to form **H**. Using any matrix inversion algorithm, we compute $\mathbf{H}^{-1}$. Since **A·H** = **S,** we have $\mathbf{A} = \mathbf{S}\cdot\mathbf{H}^{-1}$. We thus can decode all the data records in the record group. Hence, we can decode in particular our *k* data records. In contrast, we cannot perform the decoding if more than *k* data or parity records are unavailable. We would not be able to form any square matrix **H** of size *m*.

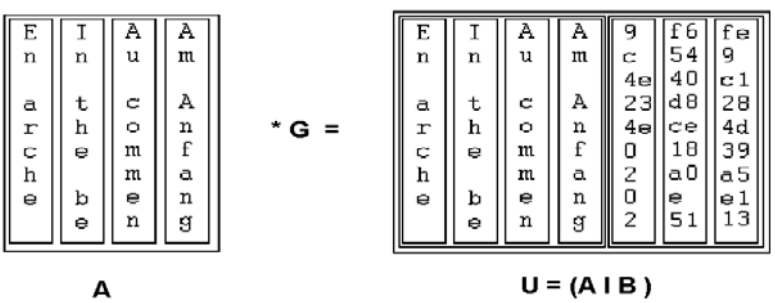

Figure 7: Definition of matrices A, B, U.

In general, if there are unavailable parity records, we can decode the data records first and then re-encode the unavailable parity records. Alternatively, we may recover these records in a single pass. We form the *recovery matrix* $\mathbf{R} = \mathbf{H}^{-1}\cdot\mathbf{G}$. Since **S** = **A·H,** we have $\mathbf{A} = \mathbf{S}\cdot\mathbf{H}^{-1}$, hence $\mathbf{U} = \mathbf{A}\cdot\mathbf{G} = \mathbf{S}\cdot\mathbf{H}^{-1}\cdot\mathbf{G} = \mathbf{S}\cdot\mathbf{R}$. Although the recovery matrix has *m* rows and *n* columns, we only need the columns of the unavailable data and parity records.

Our basic scheme in the prototype uses Gaussian elimination to compute $\mathbf{H}^{-1}$. It also decodes data buckets before recovering parity buckets. Our generic matrix **P'** has 128 rows and 129 columns for *GF* (256). As we said, to encode a group of size $m < 128$, we cut a submatrix **P** of size *m* x *m*. To apply **P'** in full is possible, but wastes storage and calculation time, since all but *m* first symbols in each line of **A** and **B** are zero. Hence the elements of **P'** other than in top left *m* x *m* submatrix would not serve any purpose.

However, the decoding according to the above scheme *a priori* requires the use and,

especially, the inversion of full size **H** derived from the *generic* generator matrix **G' = I|P'**. This despite the fact that as for the encoding using **P'**, the elements of $\mathbf{H}^{-1}$ other than those in the top left *m* x *m* submatrix of $\mathbf{H}^{-1}$ would not contribute to the result. Storing and inverting a 128 x 128 matrix is more involved than a smaller one. It would be more efficient to create the *m* x *m* submatrix **H** and invert only **H**. This requires however that the cut and the *m* by *m* inversion leads to the same submatrix $\mathbf{H}^{-1}$ as that derived by the full inversion followed by the *m* x *m* cut. Fortunately, this is the case.

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 **a** be the vector of m' symbols with the same offset in the data records of the group. The rightmost $m' - m$ coefficients of **a** are all zero. We can write **a** = (**b**|**o**), where **b** is an *m*-dimensional vector and **o** is the $m'-m$ dimensional zero vector. We split **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 **G**. Similarly; let $\mathbf{x}^l$ denote the word consisting of *m* data symbols with 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 **x** from **b.**

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

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

Here, **H** is an *m* by *m* matrix, **Y** an $m'-m$ by *m* matrix, **O** the *m* by $m' - m$ zero matrix, and **I** is the $m' - m$ by $m' - m$ identity matrix. Let $(\mathbf{x}^l|0)$ and $(\mathbf{b}^l|0)$ be the m' dimensional

vector consisting of the m coordinates of $\mathbf{x}^{l}$ and $\mathbf{b}^{l}$ respectively, and $m'-m$ zero coefficients.

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

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 **H** is invertible since **H'** is. The last equation tells us that we only need to invert the m-by-m matrix **H**. This is precisely the desired submatrix **H** cut out from the generic one. This concludes our proof.

Example 4

Consider the situation where the first three data buckets in Example 3 are unavailable. We collect the columns of **G** corresponding to the remaining four buckets in matrix:

$$\mathbf{H} = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 1 & 1a & 1c \\ 0 & 1 & 3b & 37 \\ 1 & 1 & ff & fd \end{pmatrix}.$$

We invert **H** to obtain:

$$\mathbf{H}^{-1} = \begin{pmatrix} 1 & a7 & a7 & 1 \\ 46 & 7a & 3d & 0 \\ 91 & c8 & 59 & 0 \\ d6 & b2 & 64 & 0 \end{pmatrix}.$$

The fourth column of $\mathbf{H}^{-1}$ is a unit vector, since the fourth data record is among the survivors and we need not calculate it. To reconstruct the first symbol in each data bucket simultaneously, we form vector **b** from the first symbols in the surviving buckets (D3, P0, P1, P2): **b** = (44, 5, fa, f2). This vector is the first row of the matrix **B**. We multiply $\mathbf{b}\cdot\mathbf{H}^{-1}$ and obtain (49,49,41,44), which is the first row of matrix **A**. We iterate over the lines of **B** to obtain the other rows of **A**.

4.2 Constructing a Generic Generator Matrix

We now show the construction of our generic generator matrix **G',** illustrated in Figure 9. Matrix **G** used in Example 4 above is derived from **G'**. The construction provides also

our matrix **P'** as a byproduct. Let a_j be l elements of any field. It is well known, see, e.g. [MS97], that the determinant of the l-by-l matrix that has the i^{th} power of element a_j in row i and column j is:

$$\det\left(a_j^i\right)_{0\le i,j\le l-1} = \prod_{0\le i<j\le l-1}(a_j - a_i)\,. \tag{4.1}$$

If the elements a_i are all different, then the determinant is not zero and the matrix invertible.

We start constructing **G'** by forming a matrix **V** with $n + 1$ columns and m' rows, Figure 8. The first n columns contain the successive powers of all the different elements in the Galois field $GF(n)$ starting with 0. The first column has a 1 in the first row and zeroes below. The final column consists of all zeroes but for a 1 in row $m' - 1$. **V** is the *extended Vandermonde* matrix [MS97, p.323]. It has the property that any submatrix **S** formed of m' different columns is invertible. This follows from (4.1), if **S** does not contain the last column of **V**. If **S** contains the last column of **V**, then we can apply (4.1) to the submatrix of **S** obtained by removing the last row and column of **V**. This submatrix has the determinant of **S** and is invertible, so **S** is invertible.

We transform **V** into **G'**, Figure 9, as follows. Let **U** be the m' by m' matrix formed by the leftmost m' columns of **V**. We form an intermediate matrix $\mathbf{W} = \mathbf{U}^{-1}\cdot\mathbf{V}$. The leftmost m' columns of **W** form the identity matrix, i.e. **W** has already the form $\mathbf{W} = \mathbf{I}|\mathbf{R}$. If we pick any m' columns of **W** and form a submatrix **S**, then **S** is the product $\mathbf{U}^{-1}\cdot\mathbf{T}$ with **T** the submatrix of **V** picked from the same columns as **S**. Hence, **S** is invertible. If we transform **W** by multiplying a single column or a single row by a non-zero element, we retain the property that any m' by m' submatrix of the transformed matrix is still invertible. The coefficients $w_{m',i}$ of **W** located in the leftmost column of **R** are all non-zero. If this were not the case, and $w_{m',j} = 0$ for any index j, then the submatrix formed by the first m' columns of **W** with the sole exception of column j and the leftmost column of **R** would have only zero coefficients in row j. It hence would be singular which would be a contradiction.

We now transform **W** into our generic generator matrix **G'** first by multiplying all rows j with $w_{m',j}^{-1}$. As a result of these multiplications, column m' now only contains coefficients 1? But the left m' columns no longer form the identity matrix. Hence we multiply all columns $j \in \{0,\ldots m'\text{-}1\}$ with $w_{m',j}$ to recoup the identity matrix in these columns. Third, we multiply all columns $m',\ldots n$ with the inverse of the coefficient in the first row. The resulting matrix has now also 1-entries in the first row. This is our generic

generator matrix **G'**.

We recall that the record group size for LH*$_{RS}$ is a power of 2. For the reasons already discussed for **P'**, for *GF*(256), our **I'** matrix is 128 by 128 and **P'** is 128 by 129. Hence, our **G'** is 128 by 257. Notice the absence of need to store **I'** or even **I**. We recall also that Figure 17 shows the leftmost 20 columns of **P'** produced by the algorithm using the above calculus. Likewise, Figure 14 shows a fragment of **P'** computed for *GF*(2^{16}). Finally, notice that there is no need to store even these columns. At the recovery, they can be obviously dynamically reconstructed from columns of **Q**' or **Q**, available for the encoding anyway.

$$\mathbf{V} = \begin{pmatrix} 1 & a_1^0 & a_2^0 & \cdots & a_{n-1}^0 & 0 \\ 0 & a_1^1 & a_2^1 & \cdots & a_{n-1}^1 & 0 \\ 0 & a_1^1 & a_2^2 & \cdots & a_{n-1}^2 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & a_1^{m'-1} & a_2^{m'-1} & \cdots & a_{n-1}^{m'-1} & 1 \end{pmatrix}$$

Figure 8: An extended Vandermonde matrix V with *m'* rows and $n=2^f+1$ columns.

$$\mathbf{G}' = \begin{pmatrix} 1 & 0 & \cdots & 0 & 1 & 1 & \cdots & 1 \\ 0 & 1 & \cdots & 0 & 1 & p_{1,1} & \cdots & p_{1,m'} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 1 & p_{m'-1,1} & \cdots & p_{m'-1,m'} \end{pmatrix}$$

Figure 9: Our generic generator matrix G'. Left *m'* columns form the identity matrix I. The P' matrix follows, with first column and row of ones.

5 LH*$_{RS}$ FILE MANIPULATION

The application manipulates an LH*$_{RS}$ file as an LH* file. The coordinator manages high-availability invisibly to the application. Internally, each bucket access starts in normal mode. It remains so as long as the bucket is available. Bucket availability means here that the SDDS manager at the node where the bucket resides responds to the message. If a node carrying a manipulation encounters an unavailable bucket *a*, it enters *degraded mode*. The node passes the manipulation then to the coordinator. The coordinator manages the degraded mode to possibly complete the manipulation. It initiates the *bucket* recovery operation of bucket *a*, unless it is already in progress. It performs also some other operations specific to each manipulation handled to it that we show below. Operationally, the coordinator performs in fact the requested recovery of bucket *a* as a part of the *bucket group* recovery operation. The latter recovers all

unavailable bucket(s) in the group of bucket a at once, if up to k ($k = K$, or $k = K - 1$) buckets are unavailable. For a key search, the coordinator also starts *record* recovery. This may speed up a key search by recovering the single data record or by finding that it is not in the file. If a bucket recovery already goes on, then the coordinator waits until it finishes or starts the record recovery anyway. Once the record recovery alone or the entire bucket recovery successfully terminates, the coordinator completes the requested operation.

We first present the bucket and record recovery operations. Next, we describe the application interface. The operations are file creation and removal, key search, non-key search (scan), and record insert, update or delete. Except for the file creation and removal, any of these operations may enter the degraded mode thus triggering at least a bucket recovery. Finally, we discuss the bucket split and merge that adds or removes buckets from the file, invisibly to the application. These can also enter degraded mode.

5.1 Bucket Recovery

The coordinator starts the *bucket* recovery by probing the m data and k parity buckets of the bucket group of bucket a for availability. Typically, $k = K$, unless the last change of K was not yet posted to all the groups or the group is a transitional one. The group availability level may be still then $k = K - 1$. The probe may find several unavailable buckets. If the coordinator finds up to $l \leq k$ unavailable buckets, then the failure is not catastrophic. Otherwise, the coordinator halts and reports the catastrophic failure to the user. It may still be possible to recover some records, but the case is beyond our scheme. Otherwise, the coordinator starts the *bucket group* recovery. If $l = 1$, the group recovery reduces to recovering bucket a. This should be the most frequent case. Otherwise, the operation recovers also all other unavailable data or parity buckets of the group. The coordinator first chooses a list L_A of m available buckets. The list includes the first parity bucket if it is available. The coordinator establishes also a list L_S of l spare buckets. An entry into this list contains the unavailable bucket number and the spare address. The coordinator chooses one spare as the *recovery manager*. It passes the rest of the bucket recovery task to it, with both lists. The handover prevents the coordinator from becoming a hot spot. It also avoids an additional load on a data or parity bucket that could slow down normal operations.

The manager first recreates at each spare the complete, although yet empty, structure of the bucket to be recovered there. Next, it collects the columns of **P** it needs for L_A, according to Section 4.1. It reads these from the parity buckets in L_A. It then forms matrix

H. This may include dummy columns if the group is the last one in the file and not all *m* data buckets exist yet. Then, if the first parity bucket is not the sole parity bucket to use, the manager calculates $\mathbf{H}^{-1}$. Next, it loops over the *record group* recovery that produces all the unavailable records of one group. First, it produces then all the data records. Next, - the parity records, provided there is any parity bucket number in L_S. The loop is over all the ranks of the parity records in one of the parity buckets in L_A. The manager chooses the bucket and reads one-by-one all its records. If only one data bucket is unavailable, and 1st parity bucket is in L_A, then the manager skips $\mathbf{H}^{-1}$ calculus and *GF* multiplications to recover the data bucket. We recall that the decoding is then faster, using the XORing only.

During the loop, for each parity record encountered, the manager explores its group structure field *C*, Figure 1. For every non-null key c_i, it requests the data record c_i from its bucket, provided it is in L_A (as already discussed, the bucket is also the i^{th} in the group). The manager decodes the non-key fields of unavailable data records in the group. It uses XORing for the first parity bucket, and/ or $\mathbf{H}^{-1}$. Next, using the *C*-field, it reconstructs the keys of the recovered records. Finally, if there is any parity bucket to recover, it requests the missing **Q** columns from the coordinator. It then encodes the unavailable parity records using the *m* data records in the recovered group. Finally the manager sends the recovered records (in bulks, as we discussed later) to the spares for insertion into the recovered buckets.

Once the bucket group recovery ends, the manager pushes the addresses of the recovered buckets to the existing buckets in the group so that they can update their location tables. It finally successfully returns the control to the coordinator. The coordinator considers the new buckets as ready to use. It updates accordingly its server addresses. A client or server will get the new address of a recovered bucket when it finds the bucket displaced.

It is perhaps worth recalling the alternate decoding algorithm here, for a record group with the appropriate submatrix of $\mathbf{H}^{-1}\mathbf{G}$, mentioned in Section 4.1. The resulting variant can decode all unavailable data and parity records in the group at once. Since typically only one bucket is lost, our choice is however typically faster.

As mentioned above, our bucket group recovery moves records in bulks. Formerly, we transferred them individually using UDP. This turned out to be much less effective than the current approach with the TCP/IP in passive mode (Section 6.3).

5.2 Record Recovery

The operation results from the key search for record *c*, (Section 5.4 below), that localizes it in an unavailable bucket *a*. The result for the application is nevertheless the delivery of record *c* alone, or the reply that it is not in the file. We determine the latter by consulting the *C*-field in one of the parity records in the group. More precisely, the coordinator performs record recovery in parallel with the recovery of bucket *a*. It hands control to the recovery manager at an available parity bucket of the group, giving it *c, a,* and L_A. The manager first scans the *C*-fields in the bucket for the existence of parity record (of rank) *r* with *c* in *C* (*r*). Knowing *a* speeds up this search, as only one column in *C* needs to be examined, instead of typically *m*. If the manager does not find the record, it informs the coordinator that the search for *c* is unsuccessful. Otherwise, it decodes record *c* using group *r* as described previously. Except that it calculates only record *c*, even if several data records in the group are unavailable. Finally, the manager sends the record to the coordinator.

Record recovery access time to an unavailable data record is typically much faster than the time to recover the whole bucket group (what we do anyhow). See Sections 6.3.6 and 6.3.7 below. For typically even better record recovery times, we need to avoid the intra parity bucket scan above mentioned, searching at present for key *c*. This requires an index binding rank *r* to key *c*, or an algorithm making rank *r* a function of *c* etc. The obvious trade-off is some storage and run-time overhead. The variation is a candidate future work.

5.3 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 creates also *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 **P** with the i^{th} parity bucket, with the exception of the first parity bucket (using **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 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.

5.4 Key Search

In normal mode, LH*$_{RS}$ searches for a key with the LH* key search algorithm (Section 2.1.2). The client or the forwarding server triggers degraded mode if it encounters an unavailable bucket, called a_1. It passes then 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 starts also 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 being recovered, e.g., is in another group than bucket a_1, then the coordinator starts the recovery of bucket a_2 as well. It performs than also the record recovery.

5.5 Scan

A scan returns all records in the file that satisfy a certain query Q in their non-key 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*, [LNS96]. The choice is up to the application.

5.5.1 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*$_{RS}$ applies then 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) below. 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, 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 [LNS96]. We detect unavailable buckets and enter degraded mode in the termination phase.

Example

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 has still 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.

5.5.2 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 indeed 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 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 time-out 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 5.1. 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 updates finally its image and perhaps the location data.

Example

For change, we consider now a file in state $(n, i) = (0,3)$, hence with 8 buckets 0,1…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 other $j_0 > j_4$. Likewise, condition (*ii*) cannot become true until bucket 3 replies. The client waits for further replies.

Consider now that no bucket replied within the time-out. 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.

5.6 Insert

In normal mode, an LH*$_{RS}$ client performs an insert like an LH* client. The client sends the insert request to the bucket determined by the data record key c and the client's image. The client waits for an acknowledgement to terminate. If it does not come within a timeout, then the client sends the insert to the coordinator and the operation enters degraded mode.

The receiving data bucket follows algorithm A2 (Section 2.1.2) by forwarding the request if necessary. If the correct data bucket receives the insert, it stores the record as for an LH* file. If the data bucket overflows, the bucket informs the coordinator. In addition, it assigns a rank r to the record. Next, it sends the **Δ**-record (with key) c and r to the k parity buckets. Recall that the **Δ**-record is essentially the inserted record. The data bucket then waits for the k acknowledgements from all parity buckets.

Upon the reception of the message, each parity bucket creates the parity record r if it does not already exist, and inserts c into its key-list. It also encodes the non-key data of **Δ**-record c through the update of B-field of record r. It first multiplies the Δ-record symbol-wise with the related coefficient of matrix **P**. Then, it either XORs the result to

the *B*-field already in record *r* or stores the result as the new *B*-field, if record *r* is new.

Inserts (as well as updates and delete operations) need to maintain coherency between parity records and data records. That is, a data record should be changed with all its parity records or not at all. Otherwise, the *k*-availability of all records in the group is no longer guaranteed. For instance, assume that (i) we have a group with $k = 2$, (ii) a data record was inserted, but finally only one of the parity records was updated. If the data record and one of the parity records become unavailable, then we can retrieve the inserted record if the updated parity record is still available, otherwise not. As we will see, the situation is even more difficult for updates. As the general rule, in order to maintain *k*-availability, we need to perform any insert (update / delete) operation at the data bucket and all *k* parity buckets. In other words, a change should be committed simultaneously at all buckets involved.

The commit process between the data and the parity buckets differs for $k = 1$ and $k > 1$. In the former case, we use an implicit 1-phase commit (1PC). The parity bucket simply acknowledges the reception of the Δ-record and creates/updates its record *r*. The data bucket then acknowledges the insert to the client that can eventually avert the application. The server or the client enters the degraded mode if any of the expected messages does not arrive in time.

We now discuss the case $k > 1$. 1PC no longer guarantees that all parity records and the data record are updated. We therefore use a variant of 2PC that guarantees that the **Δ**-record *c* updates all or none of the *k* buckets. The data bucket sends the **Δ**-record (with key) *c* and *r* to all *k* parity buckets. Each parity bucket starts the commit process by acknowledging the reception of the message with **Δ**-record *c* and with *r*. This confirmation constitutes the "ready-to-commit" message of 2PC. Each parity bucket encodes the record as usual into parity record *r*. But it retains the **Δ**-record in a differential file (buffer) for a possible rollback. If the data bucket gets all *k* "ready-to-commit" messages, it acknowledges to the client after it sends out "commit" message to the *k* buckets. Each bucket that receives the message discards the **Δ**-record. Notice that one could also use a more sophisticated scheme based on collective acknowledgements as in TCP/IP, but this variations is beyond our scope at present.

The degraded mode starts when any of the buckets involved cannot get a response it waits for. The data bucket enters the degraded mode if it lacks any of the acknowledgments from the parity buckets. It alerts the coordinator, transmitting *r* and the number *p* of the unavailable parity bucket (its column index in **P**). The coordinator

probes the group for the availability of *m* buckets. It also probes all the parity buckets of the group, except bucket *p*, whether they have *c* or not. In the latter case, such a bucket either has *c* in the key-field *C* and the Δ-record in the differential file, or only has *c*, or lacks both. In every case, provided the coordinator finds *m* available buckets, it synchronizes all available parity buckets so that all reflect the insert. Then, the coordinator recovers the group and finally acknowledges to the client.

Another degraded case occurs when parity bucket *p* does not receive a message from the data bucket. The data bucket must then have just failed and bucket *p* must be in the "ready-to-commit" state and must have the Δ-record. It then alerts the coordinator, sending out Δ-record *c* and *r*. The coordinator probes the bucket group for the recoverability. If the probe is successful in finding the required number of available data buckets, the coordinator synchronizes all parity buckets so that all have processed the insert. The recovery process can now proceed. The recovered data bucket will contain the inserted record. Finally, the coordinator sends an acknowledgement to the client.

Next, the client might detect that the data bucket has failed because of a lacking acknowledgement. The client informs the coordinator. After the coordinator determines the availability of buckets in the group, it synchronizes the parity buckets with regard to the insert. It might find that the data bucket never sent any messages to a parity bucket, because it failed before receiving the original insert command from the client or because it failed before it could forward the Δ-record. Alternatively it might find that all available parity buckets have already committed the insert. Otherwise, a parity bucket would have informed the coordinator of the data bucket's unavailability. In all the cases, the coordinator can determine the state because either the record key *c* is in the parity record or not. After synchronization at the parity records, all unavailable buckets in the group are recovered and the insert is finally acknowledged to the client.

Finally, we have to deal with the simultaneous unavailability of client and data bucket. In this case, either all (available) parity buckets have not received the Δ-record or all have committed the insert. Only a later file operation will discover that the data bucket is unavailable. Depending on the shared state of the parity records in regard to the insert, the data bucket will be recovered without or with the inserted record.

An insert in a degraded mode where the correct data bucket was unavailable may generate an overflow at the recovered bucket. The new bucket itself alerts the coordinator to perform a split.

5.7 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.

5.8 Update

An update operation of record *c* changes its non-key field. In the normal mode, the client performs the update as in LH*. The client sends the record with its key *c* and the new value of the non-key field. The data bucket uses *c* to look up the record, determines its rank *r*, calculates the **Δ**-record, and sends both to all k parity buckets. These recalculate the parity records. Finally, the data bucket commits the operation.

As for inserts and deletes, 1PC suffices only for $k = 1$. But for $k > 1$, 2PC as used for inserts and deletes is no longer sufficient. We cannot always make out with that protocol whether a parity record has been actually updated.

Our basic 2PC version for updates works as follows. The data bucket sends the **Δ**-record with key *c* and rank *r* to the $k > 1$ parity buckets. But now the messaging follows the order in **P**. Each parity bucket starts the commit process by acknowledging *c* and *r*. As for an insert, the acknowledgement constitutes the “ready-to-commit” message of 2PC. Each parity bucket encodes the record as usual but also keeps the **Δ**-record in its differential file. If the data bucket gets all *k* “ready-to-commit” messages, it acknowledges to the client. It also sends out “commit” messages to the *k* buckets. However, it does so one at a time, waiting for each previous acknowledgement before sending to the next (in the order in **P**) parity bucket. Once a receiving bucket gets this message, it discards the **Δ**-record.

Assume now that the coordinator is alerted because of the loss of parity bucket *p* after *p* entered the commit phase. The coordinator can find that all parity buckets before *p* have committed and that the ones after *p* have not. Therefore, the coordinator can synchronize the parity buckets accordingly.

It is easy, but tedious, to prove the correct termination for an update with this protocol for all the other cases of the degraded mode. These are as for the insert. We avoid discussing them here. Let us say only that the algorithms are quite similar, although knowing the position of the alerter may make some faster.

5.9 Split

As in LH*, if an insert to an LH*$_{RS}$ data bucket a overflows a, then a alerts the coordinator. The coordinator starts the *split* operation of bucket n, identified by the split pointer. Typically, we have $n \neq a$. In the normal mode, the coordinator first locates an available server and allocates there the new data bucket N, where N denotes the number of data buckets in the file before the split. Bucket N is usually in the bucket group different from that of bucket n, unless the file is small and $N<m$. If N is the first bucket in the group, then the coordinator allocates K new, empty parity buckets. If $K > k$ of the bucket group with bucket n, then the coordinator also allocates an additional K^{th} parity bucket to the group. Provided all this performs normally, the coordinator sends the *split* message to bucket n with all the corresponding addresses. This hands control of the split to bucket n. The coordinator waits nevertheless for the final commit message. The bucket sends all the data records that change the address when rehashed using h_{j+1} to data bucket N. We recall from Section 2.1.1 that our implementation sends these records in bulks.

For each data record that moves, bucket n finds its rank r, produces a Δ-record that is actually identical to the record itself, and requests its deletion from the parity records r in all the k buckets of its group. It also assigns new successive ranks r', starting from $r' = 1$, to the remaining data records. Bucket n sends then both ranks with each Δ-record to the K parity buckets. At the k existing buckets, it requests the delete of Δ-record from parity record r and its insert into parity record r'. A new K^{th} parity bucket, if there is one, disregards the delete requests.

At data bucket N, the bucket requests the inserts into its K parity buckets with the successive ranks it assigns. Once the split processing terminates at bucket N, N reports this to the waiting coordinator.

The operations on the parity buckets use 1PC for $K = 1$ and 2PC as described for the inserts and deletes otherwise. The degraded mode starts when a data or a parity bucket does not reply. The various cases are similar to those already discussed. Likewise, the 2PC termination algorithms are similar to those for an insert as well. We thus avoid the discussion of all these aspects of splitting here. Notice however that all unavailable

buckets are reported, as the coordinator waits for the commit messages from both buckets n and N.

Once the split terminates successfully, the coordinator resets the value of n as already described in Section 2.1.1. Notice that bucket N becomes then "officially" bucket N - 1 since $N := N + 1$.

5.10 Merge

Deletions may decrease the number of records in a bucket under an optional threshold $b' << b$, 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 rarer, we omit discussion of the algorithm for this case.

The merge updates also 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 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 are to counted are 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.

6 PERFORMANCE ANALYSIS

We now discuss the storage, communication, and processing performance of the scheme. As usual, we derive the formulae for the load factor, parity storage overhead, and the messaging costs. We discuss some design choices that appear. Next, we show the mostly

experimental analysis of the processing times. A purely formal analysis of these did not seem useful, because of the practical complexity of our system. The response times also depend heavily on various implementation level choices, as we will show.

6.1 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 $N >> m$ so that the influence of the last group is negligible. The last group contains possibly less than m data buckets,. If $K' = K - 1$, i.e., if the file is in 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 that $\alpha_g (K')$. It decreases progressively until 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 concern in practice only relatively few N values of an LH*$_{RS}$ file. The practical choice of N_1 should be indeed $N_1 >> 1$. 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 $\alpha_g (K)$. That one 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 scale-up to level $K + 1$ is incremental. One also accesses among the data buckets only to the existing splitting bucket and the new one at the time. This strategy induces a storage occupancy penalty with respect to best α_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 α_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, e.g., 64, 128, 256… Then, all k values needed in practice should remain negligible with respect to m, and $N >> 1$. 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. The penalty is however obviously about l times greater messaging cost of bucket recovery, since m buckets have to be read. It does not mean however (fortunately) that the recovery time also increases l times, as it will appear. Hence, the trade-off can be worthy in practice.

Example

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

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 thus optimal for 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 $K := 3$ etc.

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. But the 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. But 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 $K' = 3$ etc. 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 $N = 16$, 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 then the parity overhead k / m rather negligible since under 3 %.

6.2 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, and CPU performance, communication protocol, flow control strategy, bulk messaging policy etc. 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, e.g., for the synchronous update of the parity buckets. The sender considers a node unavailable if it cannot deliver its message.

Table 2 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 formulae for the dominant cost component. Their derivation is quite easy hence we only give an overview. More in depth formulae such as 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 [LNS96] 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 5.1. 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 at the average with αb records each. It dispatches the result to l-1 spares, using one message per record, since we assume reliable delivery. Here 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.

Manipulation	Normal Mode (N)	Degraded Mode (D)
Bucket Recovery (B)	$B \approx (3+2m+3k)+\alpha_d\, bm+\alpha_d\, b(l\text{-}1) + 1$	Not Applicable
Record Rec. (R)	$R \approx 2$ or $2(m\text{-}1)$	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)	$M_N = L_N$	$M_D \approx M_N + B$

Table 2: Messaging costs of an LH*$_{RS}$ file.

For the 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, $2(m\text{-}1)$ messages are typically, and at most, necessary to recover the record.

The other costs formulae are straightforward. The formulae for the insert and delete consider the use of 1PC or of 2PC. We do not provide the formulae 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 consider that *l* unavailable buckets found are each in a different group. The coefficient B_1 denotes the recovery cost of a single bucket. Several formulae can be obviously 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 order of $O(\log_{N1} N)$. A rather large N_1 should suffice in practice; at least $N_1 = 16$ most often. This dependence should thus 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 even often decrease.

6.3 Experimental Results

We have prototyped LH*$_{RS}$ to study the timing of various operations and prove the viability of the scheme. The prototype was a many-year effort. The earliest implementation is presented in [Lj00]. It put into practice the parity calculus defined in [LS00]. It also reused an LH*$_{LH}$ implementation for the data bucket management, [B02]. Experiments with next version of LH*$_{RS}$ prototyping were presented in [ML02]. The current version used for the experiments below builds upon that one. We present the prototype itself more in [LMS04]. Further details of the prototype, as well as the deeper discussion of the experiments discussed below, are in [M03].

The prototype consists of the LH*$_{RS}$ client and server nodes. These are C++ programs running under Windows 2000 Server. Internally, each client and server processes the queries and data using threads. The threads communicate through queues and other data structures and synchronize on events. There are basically two kinds of threads. The *listening threads* manage the communications at each node. There is one thread for UDP,

one for TCP/IP and one for multicast messaging. Next, four *working threads* process simultaneously the queries and data, received or send out.

The communication uses the standard UDP and TCP/IP protocols. Clients communicate with servers using UDP. A listening thread timely unloads the UDP buffers to prevent loosing a datagram. The servers communicate using TCP/IP for data transmission during the bucket split or recovery, and UDP for other needs. Again, a listening thread unloads the UDP buffers. Another such thread manages the TCP/IP stack. This stack has the listening socket in passive open mode [RFC793]. This new connection mode, available in Windows 2000, [MB00], replaced those studied in our earlier prototypes. It handles more effectively a larger number of incoming requests. It skipps indeed the connection dialogs, previously necessary for each request. It proved by far the most efficient up to now for our scheme.

We have designed the prototype to experimentally measure the speed of the operations using the parity calculus, depending on design choices. Most experiments compared the use of $GF(2^8)$ and of $GF(2^{16})$. We ourselves assumed that using the latter was faster, but could not quantify this assumption nor validate it. Experiments confirmed that using the latter was indeed usually, faster, but not always. Most noticeable speed up occurred for the decoding. We could also confirm the utility and measure the benefit of using our newest logarithmic matrix **Q**, derived from our also newest matrix **P**, with a first column and first row of ones. We then measured the speed of the operations involving the parity updating, namely the inserts, file creation with splits, updates, as well as the bucket and record recovery. The study used various availability levels, namely k = 0… 3. We left the study of deletes, of merges and of scans for the future. First two operations are of lesser practical interest. The last one is out of our goal here, as normally independent of the parity calculus. We have measured nevertheless the key search speed, as the referential of the time to operate over a data record. Each measure was averaged over several experiments.

Practical considerations lead to simplified implementation of some operations compared to their description in previous sections. Also, the experiments modified our own ideas on the best design of some operations. We discuss the differences in respective section.

The configuration for our experiments included five P4 PCs with 1.8 GHz clock rate and 512 MB memory, and a 2.6 GHz, 512 MB P4 machine. The latter was used as a client. Others were data or parity servers. Sometimes, we also used additional client machines (733 MHz, P3). Our network was a 1 Gbps Ethernet.

6.3.1 Parity Calculus Optimization

To test the efficacy of using **Q**, we conducted experiments creating parity records in a bucket with a logarithmic **Q** column, versus its original **P** column. We used the group of $m = 4$ data buckets and created a parity bucket using the second or third or fourth parity column of each matrix (the first column of **P** was that of ones). A data bucket contained 31250 records. Using $GF(2^8)$, the average processing time shrank from 1.809 sec to 1.721 sec. We saved 4.86%. Use of $GF(2^{16})$, reduced the time from 1.462 sec to 1.412 sec, i.e., by 3.42%. Notice that $GF(2^{16})$ was always faster, by about 20 %.

We next experimented similarly with **P** having the first column and the first row of ones Operationally, we used thus **Q** with first column and row of zeros. We recall that then we encode the updates to the 1st data bucket simply by XORing with the Δ-record. We wished to evaluate further speed up if any. For $GF(2^8)$, the processing shrank indeed further from 1.721 sec to 1.606 sec, i.e., by 6.68%. Using $GF(2^{16})$, we measured 1.412 sec and 1.359 sec, i.e., 3.75% of additional savings. Notice that $GF(2^{16})$ is again always faster. But slightly less this time, by about 15 % only.

Using **Q** with the first column and first row of zeros makes thus the encoding the fastest. Such **Q** was therefore our final choice for the scheme and all the experiments we report below. We attribute the better savings for $GF(2^8)$, with respect to the above percentages for $GF(2^{16})$, to the higher efficacy of the XORing for byte sized symbols.

6.3.2 Key Search

The key search time is the basic referential of access performance of the prototype, since it does not involve the parity calculus for $k > 0$. We have measured the time to perform random *individual* (synchronous) and *bulk* (asynchronous) successful key searches. All measures were at the client. The timing of an individual search starts when the client gets the key from the application. It ends when the client returns the record received from the correct server. The search time measured is the average over a synchronous series of individual searches, i.e., one after the end of another. Measuring the bulk search starts when the client gets the first key in bulk from the application. It lasts until the application gets the last record searched. Finally, we average it over the bulk size. During the bulk search, the client launches searches asynchronously, as is done usually for a database query. Our searches use UDP, hence a custom flow control method prevents a loss due to a server overload during a bulk operation. In fact, the experiments typically did not even invoke it, without showing any losses.

We have measured in this way the search times in the file of 125000 records, distributed over four buckets and servers. A record had a 4 B key and 100 B of non-key

data. The average individual and bulk search times were 0.2419 ms and 0.0563 ms respectively. Thus the former is about 40 times faster than a disk key search. The latter reaches a speed-up of almost 200 times. The server processing speed basically bounds the former. The client speed bounds the latter.

6.3.3 Insert

We have measured series of individual and bulk inserts into an empty bucket, so as to avoid a split. The timing of an individual insert starts when the client gets the record from the application. It ends when the client returns to the application the acknowledgement received from the data bucket. We only implemented the 1PC, neglecting thus in our experiments the rare case of the double failure of the data bucket and of the client. Also, in these experiments the data bucket sent the acknowledgement to the client, after sending the messages to the k parity buckets, but without waiting for the acknowledgements from these buckets. In other words, we have assumed reliable messaging. The only cause of a missing acknowledgement could be the unavailability of a parity bucket, triggering the degraded mode. The insert times measured in these conditions are the fastest possible for k–availability, except for the neglected case. Later, for the experiments with the updates, we report on the unreliable messaging performance assumption as well.

As for the experiment, we have timed the series of 10 000 inserts into an initially empty bucket of b = 10 000. We avoided any split in this way, unlike for the measure of the file creation time in Section 6.3.4 below. A record has again the 4 B key and 100 B of non-key data. The average times were in practice identical for both *GF*s used. We recorded 0.29 ms for k = 0, 0.33 ms for k = 1 and 0.36 ms for k = 2. The average bulk insert times were seven to nine times faster reaching 0.04 ms. These times were the same as for the updates, discussed in Section 6.3.5 below. They were measured in the same way, and are similarly independent of k.

The figures above show that adding the first parity bucket to 0-available file, slows down an insert on the average by 0.04 ms or 14 %. Adding one more parity bucket costs slightly less, 0.03 ms or 10 %, despite the XOR only calculus on the first bucket. The reason is that most of the operations at the data bucket are in common, and the operations at the parity buckets proceed in parallel. All this appears to be a quite efficient behavior. Finally, the measured times are respectively about 30 to 250 times faster than to local disks (assuming 10 ms per access). As for a key search, the individual insert time was bound mainly by the server speed, while the bulk insert was due to the maximal client speed.

6.3.4 File Creation

Figure 10 shows the average file creation time, by inserts with splits this time, for a bucket group of $m = 4$ data buckets and $k = 0,1,2$ parity buckets. The inserts are individual ones. We did not experiment with the bulk inserts, as they need a more complex design of splits left for future work, to prevent side effects resulting from the concurrent processing of splits and of inserts. Besides, the average time to create a file using l record bulk inserts would be at the client simply 0.04 l ms, given the bulk insert time above. At a server, the time could be somehow longer, to complete the last inserts (see the discussion of the bulk updates below). For the experiments with inserts above, during the file creation the data bucket sends the acknowledgement to the client, after sending the messages to the k parity buckets, but without waiting for the acknowledgements from these buckets. The results we measured were practically the same for $GF(2^{16})$ and $GF(2^8)$. Hence, the charts shown apply to both fields, although the numerical values shown are for $GF(2^8)$. We inserted a series of 25000 records, again with a 4 B key and 100 B of non-key data per record. The bucket size was b = 10 000. A point of the chart corresponding to l inserts shows the total time to perform these inserts.

The inserts caused the file to split thrice. The split of bucket 0 occurred naturally after the insert 10 000. A temporary slow down of the insert times resulted, greater for greater k. The next inserts went uniformly into buckets 0 and 1. After slightly more than 10 000 further inserts, both buckets split almost concurrently. That is why the chart seems to show only two splits.

From the times to insert the 25,000 records, in the figure, we can gauge the typical cost of additional parity buckets for our file, once it scales to the steady state with many groups. For k = 0, we have the creation time of 7.985 s. For k = 1, we have 10.125 s that is 27 % more. Finally, for k = 2, the time is 10.974 s that is 8 % slower than for k = 1. The related average times per record inserted were 0.32 ms, 0.41 ms, and 0.44 ms for k = 0, 1, 2 respectively. Splits introduced thus respectively the additional average costs, of 3, 8 and 8 ms, as compared to the costs of individual inserts alone. The percentage values are respectively of 10.4 %, and of 24 %. All together, these times remain nevertheless at least 20-30 times faster than to disk buckets.

As is to be expected, adding the first parity bucket causes the most noticeable degradation. The percentage value of 27 % is about twice that for an insert alone. There is indeed now the parity calculus cost also for the splits. Adding additional parity buckets has again globally much lesser effect (a 8 % slow down), also because of the parallelism of the parity updates. Notice however that there is no incidence on the cost of the updates

to the new parity bucket during the splits, as the difference to the average time per insert for $k = 2$ remains the 8 ms. It confirms logically that split processing on the parity buckets is about fully parallel. We extrapolate the increase for each value of $k > 2$ to be the same 8%. Indeed mainly the additional messaging done at the data bucket causes it.

The charts in the figure are about linear. The experiments confirm thus the scalability of the scheme, and we can predict the creation times for larger files. We create our files for $k = 0,1,2$ at the rate (speed), respectively, of 3 131, 2 469 and of 2 278 records per second. For instance, to scale up our 2-available file to 1 M records should take thus 439 s, i.e. about 7.3 m. More generally, as our records are 104 B long, we create our files at the rate of 0.33 MBs for $k = 0$, 0.25 MBs for $k = 1$, and of 0.23 MBs for $k = 2$. These numbers allow us to predict linear creation times for other record sizes. Bulk creation times and rates should be at the client and for the application yet about ten times faster. For instance thus, less than a minute should suffice for a 1 M record file.

We also timed the use of our former **Q** matrix, without the first column and row of ones. The creation time for $k = 1$ was 10.011 sec. Thus, our new **Q** effectively speeds up the encoding time, by almost 2 % here. We recall that this acceleration, although slight here, is at no other cost.

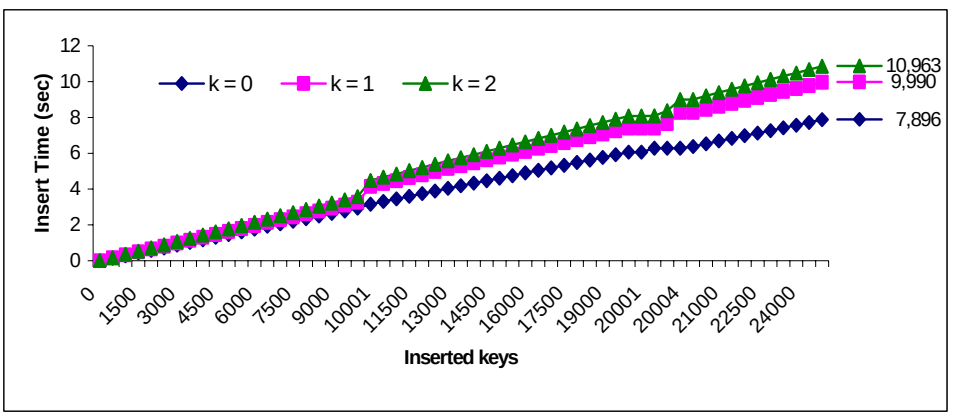

Figure 10: File creation times (seconds)

6.3.5 Update

To determine the update performance, we generated series of 500, 1000, 5000, and 8000 blind updates to the records in our LH*$_{RS}$ file (same as for the insert experiments). We updated different records, to prevent caching effects. The results are in Table 3 for bulk and individual updates. All updates are sent using UDP and 1PC. As before, we neglect the rare case of double unavailability of data bucket and of the client. This time, however, the data bucket waits for the acknowledgements before sending the commitment to the

client. The second column gives the average bulk update time in the normal mode. These measures start with the reception of the first update from the application and ends with the send-out of the last of the series. The processing at the servers may last longer. Also, if the series is longer, then more records are perhaps temporarily stored in the Listen Queue of the Listen Thread at each server. Some acknowledgements may come back to the client after the end of the bulk update. The processing time at the data bucket depends on *k*. But it does not influence the bulk update time as defined here. If any acknowledgements were negative, or missing, the client would start the degraded mode. Notice that the bulk insert time is independent of the *GF* used.

		Individual			
	Bulk	$k = 0$	$k = 1$	$k = 2$	$k = 3$
$GF(2^8)$	0.04	0.25	0.48	0.57	0.58
$GF(2^{16})$	0.04	0.24	0.50	0.55	0.59

Table 3: Average bulk and individual blind update times (milliseconds) per record.

The other columns list the average individual update times for $k = 0...3$. The bulk insert times are basically six times faster, as the comparison for $k = 0$ shows. The numbers show also that using one parity bucket doubles the insert time into the data bucket alone. This result matches the intuition. It also shows that unreliable messaging assumption costs quite a lot, when compared to the results for the experiments with the inserts. But adding more parity buckets only increases the time by 10% to 20%. Notice that adding the 3rd parity bucket adds on 2 - 7 % only. All this is again nice behavior. One may further extrapolate these results to the server side processing of the bulk updates. Finally, using $GF(2^{16})$ does not appear uniformly faster. The results are practically identical for both fields, as for inserts.

Compared also to the insert times, the bulk times do not change, as the client processes inserts and updates at the same possible speed. The update processing takes in contrast longer per record at the servers, with perhaps longer Listen Queues. This results from Table 3, as the individual update time for $k = 1$ already is almost 45 % longer than the time to insert. The individual insert time for $k = 0$ is in contrast about 15 % longer than that of an individual update. This is due to the internal LH splits within the bucket.

6.3.6 Bucket Recovery

As described in Section 5.1, the recovery manager organizes bucket recovery. For implementation related reasons, our prototype locates the recovery manager at a parity

bucket and not at a spare. To measure the performance, we simulated the creation of an LH*$_{RS}$ group with 4 data buckets and 1, 2, or 3 parity buckets. The group contained 125 000 = 4 * 31 250 data records consisting again of a 4 B key and 100 B non-key data. We then reconstructed 1, 2, and 3 "unavailable" buckets. The recovery manager loops conceptually over all the existing record groups, i.e., over all the parity records in the parity bucket (Section 5.1). In fact, it recovers records by *slices* of a given size *s*. It requests *s* successive records from each of the *m* data/parity buckets, and recovers the *s* record groups. Then, it requests next *s* records from each bucket. While waiting, it sends the recovered slice to the spare(s). Figure 11 presents the effect of slice size on the recovery of a data bucket in the sample case of using the first parity bucket with 1's only and $GF(2^{16})$. We measured the total recovery time *T*, the processing time *P*, and the communication time *C*.

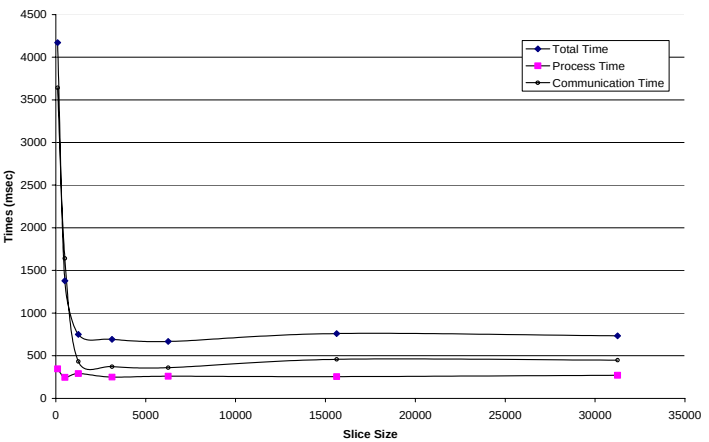

Figure 11: A single data bucket recovery time (milliseconds) as function of the slice size *s*.

The basic finding is that the recovery time greatly decreases for a larger *s*. For $s = 1$, we have $C = 149$ s, $P = 1.735$ s and $T = 165$ s. Figure 11 does not give these values since they are so large but rather displays values only for $s \geq 100$. Once *s* is above 1000, *T* drops under 1s, and *P* and *C* under 0.5 s. All the times decrease slightly for larger *s* and become constant when we choose *s* over 3000. This is a consequence of our latest communication architecture that uses the passive TCP connections we already spoke about. The result means also that a server may efficiently work with buffers much smaller than the bucket capacity *b*, e.g., 10 times smaller. The experiments with our earlier architectures are in [M03]. They prove the great superiority of the current one.

Table 4 completes Figure 11 by listing the *T, P, C* times for *s* values minimizing *T* and $k = 1,2,3$. We used $GF(2^8)$ and $GF(2^{16})$. The difference between a *T* value and the

related $P + C$ is the thread synchronization and switching time. We have measured all these times also for the other s values marked in Figure 11. For $s \geq 1250$, the differences to the times listed here were under 15 % for 1-DB recovery, 5 % for 2-DB recovery and 2 % for 3-DBs.The 1st line of the table presents the recovery of a single data bucket (1-DB), using the XOR decoding only, as at Figure 11. The second line of the table shows 1-DB recovery using the RS decoding (with the XORing and multiplications). We used another parity bucket or the first one in our initial scheme, [LS00], but not that with ones only. The XOR calculus proves notably faster for both *GF*s used. The gain was expected, but not its actual magnitude. P becomes indeed almost three times smaller for $GF(2^8)$, and almost 1.5 times smaller for $GF(2^{16})$. T decreases less, given the incidence of the C value. That value is naturally rather stable and reveals relatively important with respect to P, despite our fast 1 Gbs network. For the RS decoding we have $C > 0.5P$ at least. Even more interestingly, we reach $C > P$ for the XOR decoding.

	$GF(2^8)$				$GF(2^{16})$			
	s	***T***	***P***	***C***	***s***	***T***	***P***	***C***
1-DB (XOR)	6250	0,646	0,245	0,349	6250	0,667	0,260	0,360
1-DB (RS)	6250	1,083	0,672	0,376	6250	0,828	0,385	0,303
2-DBs	31250	1,776	1,250	0,422	31250	1,088	0,599	0,442
3-DBs	15625	2,443	1,860	0,442	15625	1,468	0,906	0,495

Table 4: Best data bucket recovery times (seconds) and slice sizes.

All together, our numbers prove the efficiency of the LH*$_{RS}$ bucket recovery mechanism. It takes only 0,667 s to recover 1 DB in our experiments, and less than 1.5 s to recover 3 DBs, i.e., 9.375 MB of data in three buckets. Notice that the growth of T appears sub-linear with respect to the number of buckets recovered. This is the consequence of the parallelism at the implementation level, and of the recovery of the bucket group as the whole, at the conceptual level. The numbers greatly contribute also to the advantage of using $GF(2^{16})$. It halves P of any recovery measured, but that using XOR only. This was the rationale for our choice of this field for the basic LH*$_{RS}$ scheme, given also its behavior for the encoding as good in practice as of $GF(2^8)$. Notice that C in Table 4 increases more moderately than T as the function of the number of DBs recovered.

The flat character of charts in Figure 11 for larger values of s confirms the scalability

of the scheme. It allows us also to guess the recovery times for larger buckets. We can infer from the above numbers that we recover a data bucket group of size m = 4 from 1-unavailability at the rate (speed) of 4.68 MBs of data. Next, we recover 2 data buckets of the group at the rate of 5.74 MBs. Finally, we recover the group from 3-unavailability at the rate of 6.38 MBs. If we thus have 1 GB of data per bucket, the figures imply T of about 3.5, 5.5 and 8 minutes, respectively. If we choose the group size m = 8, to halve the storage overhead, the recovery rates will halve as well, while the recovery time will double, etc.

Table 5 presents a single parity bucket recovery time, again for 31 250 records to recover and s = 31 250. The time T to recover the first parity bucket, using XOR only, analyzed in the line noted PB (XOR), is faster than for the other buckets using the RS calculus. We observe again fast performance. The XOR only recovery using 2B symbols, for $GF(2^{16})$, is less efficient than that using the 1B symbols, for $GF(2^8)$. The picture reverses for the other parity bucket, as the RS line in the table shows. A small difference in P value with respect to that reported in Section 6.3.1 is due to the experimental nature of the analysis. The measurements naturally slightly vary among experiments. Similarly as for the data buckets, we can infer from Table 5, the parity bucket recovery rates per MB of data stored for various values of m, and the recovery times of the parity buckets of various sizes.

	$GF(2^8)$			$GF(2^{16})$		
	T	***P***	***C***	***T***	***P***	***C***
PB (XOR)	1.872	1.316	0.317	2.062	1.484	0.322
PB (RS)	2.228	1.656	0.307	2.103	1.531	0.322

Table 5: Parity bucket recovery times (seconds) for the slice size of s = 31 250 records.

6.3.7 Record Recovery

The record recovery manager is in our prototype located at one of the parity buckets. It acts as described in Section 5.2. Table 6 shows the average total record recovery time T we have measured. The bucket size was b = 50 000. The group size was m = 4. The times are measured at the parity bucket and starts when the bucket gets the message from the coordinator, until the recovery of the record.

The times for $GF(2^{16})$ are slightly higher. The reason is that we convert 1B characters to 2B symbols and back. In any case, the average scan time of our parity bucket to locate the key c of the data record, as described in Section 5.2, was measured to be 0.822 ms.

This is the dominant part of the total time as it represents 62% and 64% respectively.

The results match the intuition and the experimental key search time. They confirm that the basic record recovery capability should be often sufficient in practice. If one seeks for faster record recovery, or buckets are much larger, the additional already mentioned index (*c*, *r*) at the parity bucket should help. For $GF(2^{16})$ and our 1st parity bucket one may estimate the decrease to almost 1.296 – 0.822 = 0.474 ms. The knowledge of the scan time, allows us to evaluate the record recovery time for other values of *m* or *b*. The communication and processing time are about linear with *m*, while the bucket scan time is linear with *b*. Notice finally that even the basic record recovery times remain significantly faster than for a disk file. In our case, the typical ratio should be about eight times at least.

$GF(2^8)$		$GF(2^{16})$	
XOR	**RS**	**XOR**	**RS**
1.285	1.308	1.297	1.327

Table 6: Record recovery times (milliseconds)

7 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 the 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 decreases substantially the total number of nodes necessary for a larger file. Finally, we consider alternate coding schemes.

7.1 Parity Messaging

Often, an update changes only a small part of the existing data record. This is for instance the case of 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 alternate design is to send the Δ-record only to the first parity bucket, and without the 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 is that now updating needs two rounds of messages. The advantage is simpler parity management at the data buckets. The 1 PC suffices for the dialog between the data bucket and the first parity bucket. The management of the ranks becomes also transparent to the data buckets, as well as of 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 l^{th} 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 the lack of the commit message, the coordinator probes each other available parity bucket for its s_l. The parity update was done iff any bucket *p'* probed had $s_l{}^{p'} \neq s_l{}^{p}$. Recall that the update had to be posted to all or none of the available parity buckets that were 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 as the data bucket may send messages in parallel. The disadvantage is an additional field to manage, necessary for updates only.

7.2 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 the 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 k^{th} 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 every alternate coding scheme we discuss below. 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.

Vice versa, to add a parity bucket to a group requires a new node for it with (k + 1) column of **Q** (or **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 in parallel the data buckets. Their efficiency remains to be studied. As above, it is easy to set up the operation in parallel for all the groups in the file. Also as above, the existing parity records do not need the recalculation, unlike for other candidate coding schemes for LH*$_{RS}$ we investigate below.

Adding a parity bucket operation can be concurrent with normal data bucket updates. Some synchronization is however necessary over 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 the result they will start sending there the Δ-record for each update coming afterwards. Next, the new bucket may create its parity records in their rank order. The bucket encodes then 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, requesting then also the Δ-record.

7.3 Group Size Tuning

We recall that the group size m for LH*$_{RS}$ is basically 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 have each k parity buckets. Seldom, if the split pointer is in the left group, and the file is changing its availability level, the right group may have an availability level of k-1. 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 **P** and **Q** for the parity buckets other than the first one is however now provided with 2m elements, instead of top m only 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 the 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 this 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.

The group size halving splits in contrast 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, i.e., its key is removed from the key list of its parity records and its non-key data are XORed to the B-fields of these records. In the same time, it is encoded into the new parity buckets as if it was just inserted into the file. Again, there is a number of ways to implement the group size halving efficiently that remain open for study.

7.4 Parity Load Balancing

In the basic scheme, the data and parity buckets are at separate nodes. A parity bucket sustains also the updating processing load up to m times that of the update load of 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, e.g., on, let us say, 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 the additional nodes.

The following variant decreases the storage and processing load of the parity records on the node 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 i^{th} parity record in the record group with rank r, i = 0,1…k - 1. Assume that for each (data) bucket group there is a parity bucket group of m buckets, numbered 0,1…m – 1, of capacity kb/m records each. Store each parity record in parity bucket $j = (r + i) \bmod m$. Does it as the primary record, or an overflow one if needed, as usual. Place the m parity buckets of the first group, i.e., containing data buckets 0,…, m-1, on

the nodes of the data buckets of its immediately right group, i.e., with data buckets $m,...,2m-1$. 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 loss access to a single parity record per group. This is the key requirement to the *k*-availability, as for the basic scheme. The LH*$_{RS}$ file remains consequently *K* available. The parity storage overhead, i.e., 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 becomes also 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, when the last group is the left one, and the last file bucket $N - 1$ is its last one. Vice versa, when this bucket is the last in a right group, this overhead is zero. On the 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 the average. In other words, it reduces from 12.5 % to less than 1 % at worst. For our $N = 1$K 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 example above, 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 etc. 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, of partitioning of 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 the factor possibly about reaching l/n. The overall recovery

time possibly improves as well. The gain may be substantial for large buckets and $n >> 1$.

7.5 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 the interest in higher availability RAID [H&a94, SB96,BM93, BBM93, BBM95, B&al95, SS96]. Proposals for high availability storage systems [CMST03,X&a03] (encompassing thousands of disks for applications such as large email servers [Ma02]), massive downloads over the WWW [BLMR98,BLM99], and globally distributed storage [AK02],[WK02] maintain the constant interest in new erasure correcting codes. These may compare favorably with generalized Reed-Solomon codes. 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, etc.

We will now discuss therefore replacing our code with other erasure correcting codes, within the scope of our scheme. Certain codes allow to trade-off 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.

7.5.1 Design Properties of an Erasure Correcting Code for LH$_{RS}$*

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 (2) that also allows us to compress the delta record by only transmitting non-zero 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 failure) 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 result also in XORing because of the row of ones in the parity matrix. Finally, we can use the logarithmic matrices.

Our performance results (Section 6.3) show, that the update performance at the second, third, etc. parity bucket is therefore adequate. We recall that for $GF(2^{16})$, the slow down was of 10 % for the 2nd parity bucket and of additional 7 % for the 3rd one, with respect to the 1st bucket only, Table 3. It is further 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 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).

7.5.2 Candidate Codes

Array Codes

These 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 [PB76], [Pat85], [FHB89]. Figure 12 shows an example with m = 3 data records and k = 3 parity records. The data records form the three leftmost columns, that is, $a_0,a_1,...$, $b_0,b_1,...$, $c_0,c_1,...$ 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, etc. 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, i.e., 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 respectively 2 and 4 symbols. In general, parity record or column K has an overhang of $K(m\text{-}1)$ symbols. A group with k parity records and m data records of length L has a combined overhang of $k(k\text{-}1)(m-1)/2$ symbols, so that the parity overhead comes

to $\frac{k}{m} + \frac{k(k-1)(m-1)}{2mL}$ symbols. The first addend here 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 for that at different positions in each parity record, Figure 12. The updates proceed at the fastest possible speed for all data and parity buckets. Unlike in our case where 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 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 reduce the parity overhead by using also negative slopes, at the added expense of the decoding complexity (inversion of a matrix in the field of Laurent series over $GF(2)$).

					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	$a_5 \oplus b_5 \oplus c_5$	$a_7 \oplus b_6 \oplus c_5$	$a_9 \oplus b_7 \oplus c_5$

Figure 12: Convolutional array code.

Block array codes are another type of codes that are MDS. They avoid indeed the overhang in the parity records. As an example, we sketch the code family $B_k(p)$, [BFT98], where k is the availability level and p is a prime, corresponding to our m, i.e., $p \geq k + m$. Prime p is not a restriction, since we may introduce dummy symbols and data records.

In Figure 12 for instance, a_i, b_i, c_i with $i > 5$ at are dummy symbols. Next, in Figure

13 we have chosen $k = 2$ and $m = 3$, hence $p = 5$. We encode first four symbols from three data records a_0, a_1,..., b_0, b_1,..., and c_0,c_1,... The pattern repeats for 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 13 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 however, 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 13 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,… 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 to 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, etc. In our example, increasing k to 3 involves setting p to 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, [BFT98]. All things considered, using $B_k(p)$ does not seems a good choice for $k > 2$.

The *EvenOdd* code, [BBM93, BBBM95, BFT98], 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 used. The experimental analysis in [S03] 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 [C&al04].

EvenOdd can be generalized to $k > 2$, [BFT98]. 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* [XB99]. 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.

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

Figure 13: A codeword of $B_2(5)$ with a fictional row of zeroes added.

Low density parity check codes

*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, [AEL95, BG96, G63, MN97, MN99, Ma00]. We can use bits, bytes, words, or larger bit strings as symbols. The *weights*, i.e. the number of ones in a parity matrix column or row are always small, and often a constant. Recent research (e.g. [LMSSS97, CMST03]) 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, [Ma00]. LDPC codes are not MDS, but good ones come close to being MDS. Speed and closeness to MDS improve as the matrix size increase. The Digital Fountain project used a Tornado (LDPC) code for $M =$ 16000, with 11% additional storage overhead at most, [BLMR98]. [Ma00] 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 data unit size and use chunks of M/m bytes per data bucket. Each block of M bytes is distributed so the i^{th} chunk is in i^{th} 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, as for the record groups using our coding. This choice also increases recovery cost. However, the choice of the *m* value is less critical here, 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 with zeros the last bytes. One can use Δ-records calculated over the chunk(s) of the updated data record to send updates to the parity buckets as LDPC codes are linear.

If application data records consist of hundreds of bytes or are 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 in our code however, an update will usually change then 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, .e.g., the 11% quoted above. 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, close to the fastest possible, i.e., of the symbol-wise XORing of the data and parity symbols, like for the first parity record of our coding scheme, [BLM99]. Notice however, that encoding and decoding are only part of the processing cost in LH*$_{RS}$ parity management. The figures in Section 6.3 show that the difference in processing using only the 1st parity bucket and the others is by far 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, e.g., [H&al94], [CCL00], [CC01. 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, e.g., 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 non-systematic RS code in order to speed up the matrix inversion during decoding. The Kohinoor project, [M02], developed a specialized RS code for group size n = 257 and k = 3 for a large disk array to support an email server. [P97] seemed to give a simpler and longer (and hence better) generator matrix for an RS code, but [PD03] retracts this statement.

8 RELATED WORK

Traditionally, in both the centralized and distributed environment, high availability was not part of a (key-based) data structure. If needed, a lower storage level provided it such as mirroring or a RAID like technique. This approach simplifies the design of a data structure. It can in contrast deteriorate access times in the distributed environment. For example, a dictionary data structure using hashing could place a data unit at some particular node. But the underlying RAID system could replace the data at a different node or even distributes it over several nodes. This lower level interference would results in additional messaging that an integration of the parity data management into the hashing structure could avoid.

The problem is more acute for a scalable distributed storage environment with a large number of nodes. The elementary reliability calculus shows that higher levels of availability are often necessary for a data structure stored on many nodes. One approach provides the high level at each node. This approach fails if the storage nodes are standard PCs or workstations, especially in a P2P network where nodes may have low availability [WK02]. In addition, files in the same environment may require different availability levels just because of their different sizes. The alternative is to integrate high availability into scalable distributed data structures and let the availability level itself scale.

In response to the need of integrating high-availability and SDDS the concept of a *high-availability data structure* appeared, [LN96]. The first high-availability SDDS was LH*$_{M}$, where high-availability results from mirroring two LH* files. The files contain exactly the same records. They may however differ by the internal structures, e.g., the bucket size. In any case, the two files in LH*$_{M}$ are more strongly coupled than usual mirrors. This resolves some double and more failures that would be otherwise catastrophic.

[L&al97] proposed another 1-availability SDDS called LH*$_{S}$. Here, one partitions a record into n segments, stored each at a different site. There is also the (n+ 1) XOR parity segment at some other site. Compared with LH*$_{M}$, the parity overhead is much smaller,

namely close to 1/*n*. The operations require in contrast more messages. A LH*$_S$ key search in normal mode needs *n* messages, even though the messages are shorter.

Another 1-available SDDS, LH*$_g$, [LR97, L97, LR01] keeps records intact. It introduces the concept of record groups used by LH*$_{RS}$. Retrospectively, the LH*$_{RS}$ parity calculus could generalize LH*$_g$ to higher availability. As for LH*$_{RS}$, an LH*$_g$ record enters a record group when it is created. The group members are always on different servers and the group contains an additional parity record of the same structure as a LH*$_{RS}$ parity record. The initial record group is the same for an LH*$_g$ record as for an LH*$_{RS}$ record. However, an LH*$_g$ record keeps its initial record group membership, regardless of its moves caused by splits. In comparison to LH*$_{RS}$, LH*$_g$ splits are then faster. In contrast, a data bucket recovery processing is more costly. In particular, one always scans all the parity buckets, instead of usually one only for LH*$_{RS}$. Notice that the recovery is not then necessarily longer than for LH*$_{RS}$, as the scans can be parallel. If the communication is slow with respect to the processing time, it can be even faster.

LH*$_{SA}$ was the first SDDS to achieve scalable availability, [LMR98], [LMRS99]. To achieve *k*-availability, LH*$_{SA}$ places each records in *k* or *k*+1 different record groups that only intersect in this one record. Each record group has an additional parity record, basically consisting of the XOR of the other records. LH*$_{SA}$ places the buckets conceptually into a high-dimensional cube with *n* buckets in the first *k* or *k*+1 dimensions. Just as for LH*$_{RS}$, a controlled or an uncontrolled strategy adds parity buckets. A small LH*$_{SA}$ file with a $k > 1$ has a larger storage overhead than a corresponding LH*$_{RS}$ file. This advantage of LH*$_{RS}$ dissipates however for larger files. LH*$_{SA}$ parity calculations use only XORing, which gives it an advantage over *k*-available LH*$_{RS}$ files for $k > 1$. However, if there is more than one unavailable bucket, recovering a lost record can involve additional recovery steps. Deeper comparison of trade-offs between LH*$_{SA}$ and LH*$_{RS}$ remains to be explored.

Outside the domain of SDDSs, research has addressed high-availability needs for distributed flat files for many years. The dominant approach was the replication, [H96]. The major issue was the replicas consistency, [P93]. Disk arrays in a centralized environment needed historically high availability with less storage overhead [BM93], [H&a94]. The arrays have typically a fixed number of disks so that the proposed high-availability schemes were static. The aspects under investigation were mainly the parity update mechanisms (e.g. parity logging), and the parity placement providing the 1-availability through XORing. These were the performance determinants of a disk array. Next, parity placement schemes appeared intended for larger, but still static, arrays, e.g.,

[ABC97]. Current research increasingly focuses on very large storage systems, using an expandable number of storage units, whether disks or entire servers. Recent proposals for the $k > 1$ k-available erasure correcting codes already discussed in Section 7.5 came in this context.

High-availability is also a general goal for a DBMS. Nevertheless, our facet of this concept, concerning the unavailability of a part of data storage, received relatively little attention. The general assumption seems to be the use of a high-availability storage or file system underneath. Typically, it should be a software or hardware RAID storage. For a parallel DBMS, this should concern each DBMS node. At the database layer, the replication seems the only technique used. The DBMS is then typically 1-available, with respect to storage node unavailability.

The Clustra DBMS, now a commercial product, proposes a DBMS level structure that some claim the most efficient in the domain, [S99]. It hashes partitions a table into fragments located each at a different node. The nodes communicate using a dedicated high-speed switch. The Clustra hashing is static, hence with limited scalability compared to LH*$_{RS}$. The practical limit is 24 nodes at present. Each fragment is replicated on two nodes, using the primary copy approach. If a fragment is unavailable, (detected by lack of heart beat basically), its available copy, possible the primary one, is copied to a spare. The partitioning limits the recovery to a single fragment typically. The whole scheme makes Clustra tables only 1-available and limits its scalability compared to our scheme. The conclusion holds for other prominent DBMSs, whether they use for the parallel table partitioning the (static) hashing (DB2), or range partitioning (SQL Server) or both (Oracle).

Research also starts addressing the high-availability needs of scalable disk farms, [X&al03], [X&al04]. These should be soon necessary for the grid computing, and very large Internet databases. Some simple techniques are already in everyday use. They are apparently replication based, but covered by the corporate secret. The prominent example is Google. The gray literature estimates its farm spreading already over more than 10 000 Linux nodes, perhaps as much as 54 000, [D03], [E03]. There are also open research proposals for high-availability distributed data structures over large clusters specifically intended for the Internet access. One is a distributed hash table scheme with built-in specific replication, [G00]. An on-going research project follows up with the goal of a scalable distributed highly-available linked B-tree, [B03].

Emerging P2P applications, including the Wi-Fi ones, also lead to compelling high-availability storage needs, [AK02], [K03], [D00]. In this new environment the

availability of the nodes should be more “chaotic” than one typically supposed in the past. Their number and geographical spread-out should often be also orders of magnitude larger. Possibly easily running in near future into hundreds of thousands and soon reaching millions, spread worldwide. This thinking clearly shares some rationales for LH*$_{RS}$. Our scheme could thus reveal useful for these new applications as well.

9 CONCLUSION

LH*$_{RS}$ is a high-availability scalable distributed data structure. It scales up to any size and any availability level k one can reasonably foresee for an application these days. The file scalability is transparent to the application, as for any SDDS. The k-availability may scale transparently as well, or may be adjusted by the application on demand.

The scheme matured in many aspects with respect to our initial proposals, [LS00]. The evolution concerned the parity calculus, and various algorithmic issues to make the file always at least $(K - 1)$ – available, and the parity calculus the fastest. We thus have increased the Galois Field size to $GF(2^{16})$. We have evolved the parity matrix **P** so it has 1st column and row of 1s. We have also improved the calculus so to take advantage of the logarithmic parity. We have built a prototype implementation, proving the feasibility of the scheme. We have experimented on this basis with the new, and the former, parity calculus, as well as with the above-mentioned algorithmic issues. Performance analysis proved substantial speed-up of various operations.

At present, for the most frequent case of $k = 1$, the scheme performs as well as any popular 1-available RAID scheme using XORing only. For $k > 1$, it appears more effective in practice than if we used any alternate parity code or scheme we are aware of. This concerns our own earlier approach, as we just mentioned. The yet unique presence of the row of 1’s contributes to this performance. In particular, while the parity storage and communication overhead increase substantially with k used, they globally always remain close to the optimal bounds. Another known high-availability SDDS scheme may nevertheless eventually outperform the LH*$_{RS}$ on a selected feature. The diversity should profit the applications.

“Au finale”, the experimental performance analysis has shown very fast access and recovery performance. Our testbed files with 125K records, recovered in less than a second from a single unavailability and in about two seconds from a triple one. Individual search, insert and update times were at most 0.5 msec for a 3-available file. Bulk operations were several times faster. This performance is also due to the data processing in the distributed RAM. All together, the capabilities of our scheme should attract numerous applications, including the exciting new ones in the domains of grid

computing and of P2P. In particular, they should be useful for DBMSs. Those still use for the high-availability the more limited static and 1-available replication or RAID storage schemes.

Future work should concern experiments with applications of our scheme. One should also port the parity subsystem to other known 0-available SDDS schemes. The range partitioning schemes appear preferential candidates. One should also add the capabilities of the concurrent and transactional access to LH*$_{RS}$. Notice that the data records of a record group conflict on the parity records. One should finally study more in depth the outlined variants.

REFERENCES

[ABC97] ALVAREZ, G., BURKHARD, W., CRISTIAN, F. 1997. Tolerating multiple failures in RAID Architecture with Optimal Storage and Uniform Declustering, In *Intl. Symposium on Comp. Architecture, ISCA-97*.

[AEL95] ALON, N., EDMONDS, J., LUBY, M. 1995. Linear time erasure codes with nearly Optimal Recovery. In *Proc. 36th Symposium on Foundations of Computer Science (FOCS)*.

[AK02] ANDERSON, D., KUBIATOWICZ, J. March 2002. The Worldwide Computer. In *Scientific American*, 286, no. 3, March 2002.

[B03] The Boxwood Project. http://research.microsoft.com/research/sv/Boxwood/.

[B&al95] BLOMER, J., KALFANE, M., KARP, R., KARPINSKI, M., LUBY, M., ZUCKERMAN, D. An XOR-based erasure-resilient coding scheme, Tech. Rep., Intl. Comp. Sc. Institute, (ICSI), UC Berkeley, 1995.

[B99] BARTALOS, G. Internet: D-day at eBay. *Yahoo INDIVIDUAL INVESTOR ONLINE*, (July 19, 1999).

[B99a] BERTINO, E., OOI, B.C., SACKS-DAVIS, R., TAN, K.L., ZOBEL, J., SHIDLOVSKY, B., CATANIA, B. 1999. Indexing Techniques for Advanced Database Systems. Kluver. 1999.

[BDNL00] BENNOUR, F, DIÈNE, A., NDIAYE, Y. LITWIN, W. 2000. Scalable and distributed linear hashing LH*$_{LH}$ under Windows NT. In *SCI-2000 (Systemics, Cybernetics, and Informatics),* Orlando, Florida.

[B02] BENNOUR, F. 2002. Performance of the SDDS LH*$_{LH}$ under SDDS-2000. In *Distributed Data and Structures 4 (Proc. of WDAS 2002)*, p. 1-12, Carleton Scientific.

[BBM93] BLAUM, M., BRUCK, J. MENON, J. 1993. Evenodd: an optimal scheme for tolerating double disk failures in RAID architectures". IBM Comp. Sc. Res. Rep. vol. 11.

[BBM95] BLAUM, M., BRADY, J., BRUCK, J. MENON. 1995. EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures. In *IEEE Trans. Computers*, vol. 44, p. 192-202.

[BFT98] BLAUM, M., FARRELL, P.G., VAN TILBORG, H. 1998. Array Codes, Chapter 22 in *Handbook of Coding Theory*, vol. 2, PLESS, V.S., HUFFMAN, W.C. Eds. Elsevier Science B.V.

[BG96] BERROU, C. and GLAVIEUX, A. 1996. Near optimum error correcting coding and decoding: Turbo-codes. In *IEEE Transactions on Communications*, vol. 44, p. 1064-1070.

[BM93] BURKHARD, W.A. and MENON, J. 1993. Disk array storage system reliability. In *Proc. 22nd Intl. Symp. on Fault Tolerant Computing*, Toulouse, p. 432-441.

[BLMR98] BYERS, J. LUBY, M., MITZENMACHER, M., REGE, M. 1998. A digital fountain approach to reliable distribution of Bulk Data, Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM 98), Vancouver, B.C. p. 56-68.

[BLM99] BYERS, J. LUBY, M., MITZENMACHER, M. 1999. Accessing multiple sites in parallel: using tornado codes to speed up downloads. In *Proc. of IEEE INFOCOM*, p. 275-283.

[BVW96] BREITBART, Y., VINGRALEK, R., WEIKUM, G. 1996. Load control in scalable distributed file structures. *Distributed and Parallel Databases*, Vol 4(4), p. 319-354.

[BV98] BREITBART, Y., VINGRALEK, R. Addressing and balancing issues in distributed B+ trees. In *1st Workshop on Distributed Data and Structures (WDAS '98)*, Carleton-Scientific, 1998.

[CACM97] Special Issue on High-Performance Computing. Communications of ACM. (Oct. 1997).
[CCL00] CHEE, Y., COLBOURN, C., Ling, A. 2000 Asymptotically optimal erasure-resilient codes for large disk arrays. *Discrete Applied Mathematics*, vol. 102. p. 3-36.
[CC01] COHEN, M. C., COLBOURN, C.. Optimal and pessimal orderings of Steiner triple systems in disk arrays. Latin American Theoretical Informatics. p. 95-104. 2001
[CERIA] CERIA Home page: http://ceria.dauphine.fr/
[CMST03] COOLEY, J., MINEWEASER, J., SERVI, L., TSUNG, E. 2003. Software-based erasure codes for scalable distributed storage. In *USENIX Association Proceedings of the FAST 2003 Conference on File and Storage Technologies* (FAST 2003).
[C&al04] 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 Third Usenix Conf. on File and Storage Technologies (FAST'04). San Francisco, p. 1-14.
[CRP06] http://www.contingencyplanningresearch.com/cod.htm Cost of a downtime Study 1996.
[D00] DINGLEDINE, R., FREEDMAN, M., MOLNAR, D. The Free Haven Project: Distributed Anonymous Storage Service. *Workshop on Design Issues in Anonymity and Unobservability*, July 2000.
[D03] Donoghue, A. Boldly Googling into the future.
http://insight.zdnet.co.uk/internet/ecommerce/0,39020454,39116781,00.htm
[E03] Moving up the stack. www.economist.com. May 14, 2003.
[FHB89] FUJA, T., HEEGARD, C., BLAUM, M. 1986. Cross parity check convolutional codes. In *IEEE Transactions on Information Theory*, vol. IT-35, p. 1264-1276.
[G63] GALLAGER, R.G. *Low-Density Parity Check Codes***.** Nr. 21 Research Monograph Series, MIT Press, 1963.
[G00] GRIBBLE, S., BREWER, E., A., HELLERSTEIN, J., & CULLER, D. Scalable, Distributed Data Structures for Internet Service Construction. 4th Symp. on Operating Systems Design and Implementation (OSDI 2000).
[H96] HASKIN, R., SCHMUCK, F. 1996. The Tiger Shark File System. COMPCON-96, 1996.
[H&a94] HELLERSTEIN, L., GIBSON, G., KARP, R., KATZ, R., PATTERSON, R. 1994. Coding techniques for handling failures in large disk arrays. Algorithmica, vol. 12, p. 182-208.
[K98v3] KNUTH, D. *The Art of Computer Programming. Vol.* 3 *Sorting and Searching*. 2nd Ed. Addison-Wesley, 1998, 780.
[K03] KUBIATOWICZ, J. Extracting Guarantees from Chaos. In *Communications of the ACM*, 46, 2, Feb. 2003.
[KLR96] KARLSON, J., LITWIN, W., RISCH, T. 1996. LH*$_{LH}$: A scalable high performance data structure for switched multicomputers. In APERS, P., GARDARIN, G., BOUZEGHOUB, M., (eds.) *Extending Database Technology*, EDBT96*,* Lecture Notes in Computer Science, vol. 1057. Springer Verlag.
[L97] LINDBERG, R. 1997. 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.
[L80a] LITWIN, W. Linear Hashing: A New Tool for File and Table Addressing. Reprinted from *VLDB80* in Readings in Databases**,** edited by M. Stonebraker, 2nd Edition, Morgan Kaufmann Publishers, 1994.
[L80b] LITWIN, W. Linear Hashing: a new algorithm for files and tables addressing. Intl. Conf. on Databases. Aberdeen, Heyden, p. 260-275. 1980.
[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.
[LMR98] LITWIN, W., MENON J., RISCH, T. LH* with Scalable Availability. IBM Almaden Res. Rep. RJ 10121 (91937), (May 1998).
[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, Princeton U. Carleton Scientific, 1999.
[LMS04] LITWIN, W., MOUSSA, R., SCHWARZ, T., LH*$_{RS}$: A Highly Available Distributed Data Storage System. Res. Prototype Demonstration. VLDB Toronto 2004.
[LNS93] LITWIN, W., NEIMAT, M.-A., SCHNEIDER, D. Linear Hashing for Distributed Files. ACM-SIGMOD International Conference on Management of Data, 1993.

[LNS96] LITWIN, W., NEIMAT, M.-A., SCHNEIDER, D. A Scalable Distributed Data Structure. ACM Transactions on Database Systems (ACM-TODS), Dec. 1996.
[LN96] LITWIN, W., NEIMAT, M.-A. High-Availability LH* Schemes with Mirroring", Intl. Conf. on Cooperating Information 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).
[LR01] LITWIN, W., RISCH, T. LH*g: a high-availability scalable distributed data structure by record grouping. IEEE Transactions on Knowledge and Data Engineering, vol. 14(4), p. 923-927, 2001.
[LS00] LITWIN, W., SCHWARZ, TH. LH*$_{RS}$: A High-Availability Scalable Distributed Data Structure using Reed-Solomon Codes. ACM-SIGMOD International Conference on Management of Data, 2000.
[LR02] LITWIN, W., RISCH, T. LH*g : a High-availability Scalable Distributed Data Structure by Record Grouping. IEEE Transactions on Knowledge and Data Engineering, 14, 4, July/Aug. 2002.
[Lj00] LJUNGSTRÖM, M.: Implementing LH*$_{RS}$: a Scalable Distributed Highly-Available Data Structure, Master Thesis, Feb. 2000, CS Dep. U. Linkoping, Sweden.
[LMSSS97] LUBY, M., MITZENMACHER, M., SHOKROLLAHI, M., SPIELMAN, D., STEMANN, V. Practical Loss-Resilient Codes, STOC 97, Proceedings of the twenty-ninth annual ACM symposium on Theory of Computing, El Paso, TX, 1997, p. 150-159.
[Ma00] DAVID MACKAY: Home Page http:// www.inference.phy.cam.ac.uk / mackay / CodesFiles.
[Ma02] MANASSE, M. Simplified construction of erasure codes for three or fewer erasures. Tech. report kohinoor project, Microsoft Research, Silicon Valley. 2002. http://research.microsoft.com/ research/sv/Kohinoor/ReedSolomon3. htm.
[MB00] D. MACDONAL, W. BARKLEY. MS Windows 2000 TCP/IP Implementation Details. http://secinf.net/info/nt/2000ip/tcpipimp.html.
[MN97] MACKAY, D.J.C., NEAL, R.M.: Near Shannon limit performance of low density parity check codes. Electronic Letters. Vol. 33(6), p. 457-458. 1997.
[MN99] MACKAY, D.J.C., NEAL, R.M.: Good Error Correcting Codes Based on Very Sparse Matrices, IEEE Transactions on Information Theory, vol. 45(2), 1999, p. 399-431. See also http://wol.ra.phy.cam.ac.uk/mackay.
[MS97] MACWILLIAMS, F. J., SLOANE, N. J. A. The Theory of Error Correcting Codes. Elsevier/North Holland, Amsterdam, 1997.
[ML02] R. MOUSSA & W. LITWIN. Experimental Performance Management of LH*$_{RS}$ Parity Management, WDAS 2002 proceedings Carleton scientific, pp. 87-98.
[M03] MOUSSA, R. Experimental Performance Analysis of LH*$_{RS}$. CERIA Res. Rep. June 2003 [CERIA].
[Pat85] PATEL, A. M. Adaptive cross parity code for a high density magnetic tape subsystem". In *IBM Journal of Research and Development*. Vol. 29, p. 546-562, 1985.
[P93] PÂRIS, J.F. The management of replicated data. In *Proceedings of the Workshop on Hardware and Software Architectures for Fault Tolerance*. Mt. St. Michel, Fr. June 1993.
[P97] PLANK, J. 1997. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. In *Software – Practice and Experience*. Vol. 27(9), p. 995-1012. (See below).
[PD03] PLANK, J. and DING, Y. Correction to the 1997 tutorial on Reed-Solomon coding. Tech. Report, UT-CS-03-504, University of Tennessee, April 2003. (submitted)
[PB76] PRUSINKIEWICZ, P. BUDKOWSKI, S. A double-track error-correction code for magnetic tape. *IEEE Transactions Computers,* vol. C-19. p. 642-645. 1976.
[R98] RAMAKRISHNAN, K. *Database Management Systems*. McGraw Hill, 1998.
[R89] RABIN, M., O. Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance. JACM, 36, April 1989 335-348.
[RFC793] RFC 793 - Transmission Control Protocol (TCP) – Specification. Information Sciences Institute, Sept. 1981, http://www.faqs.org/rfcs/rfc793.html http://www.faqs.org/rfcs/rfc793.html
[S99] SABARATNAM M., . TORBJORNSEN, HVASSHOVD, S.-O.. Evaluating the effectiveness of fault tolerance in replicated database management systems. *29th. Ann. Int'l Symp. on Fault Tolerant Computing, 1999.*
[S03] SCHWARZ, T. Generalized Reed-Solomon Codes for Erasure Correction in SDDS.

Workshop on Distributed Data and Structure 4, WDAS-4, Paris. Carleton Scientific 2003.
[SS96] SCHWABE, E. J., SUTHERLAND, I. M. Flexible usage of redundancy in disk arrays. *8th ACM Symposium on Parallel Algorithms and Architectures, p.99-108, 1996.*
[SB96] SCHWARZ, T., BURKHARD, W. Reliability and performance of RAIDs. *IEEE Transactions on Magnetics*, vol. 31, 1996, 1161-1166.
[SDDS] SDDS-bibliography. http://192.134.119.81/SDDS-bibliograhie.html, http://ceria.dauphine.fr/witold.html
[VBW98] VINGRALEK, R., BREITBART, Y., WEIKUM, G. SNOWBALL: Scalable storage on networks of workstations. *Distributed and Parallel Databases*, vol. 6(2), p. 117-156, 1998.
[WK02] WEATHERSPOON, H., KUBIATOWICZ, J. Erasure coding vs. replication: a quantitative comparison. *1st Intl. Workshop on Peer-to-Peer systems, IPTPS-2002*. March 2002.
[X&al03] XIN, Q., MILLER, E., SCHWARZ, T., BRANDT, S., LONG, D., LITWIN, W. 2003. Reliability mechanisms for very large storage systems. *20th IEEE mass storage systems and technologies (MSST 2003),* p. 146-156. San Diego, CA.
[X&al04] XIN, Q., MILLER, E., and SCHWARZ, T. 2004. Evaluation of distributed recovery in large-scale storage systems. In Thirteenth IEEE Intern. Symp. on High Performance Distributed Computing (HPDC'04), Honolulu, HI.
[XB99] XU, L., BRUCK, J. X-code: MDS array codes with optimal encoding. *IEEE Trans. on Information Theory*, Vol. 45(1), Dec. 1999.

APPENDIX A PARITY MATRICES

We present first 32 rows by 10 columns submatrices of the generic parity matrix **P'** and of the generic logarithmic parity matrix **Q'** for $GF(2^{16})$ we use for LH*$_{RS}$. The values are four hexadecimal digits. The submatrices allow for actual matrices **P** and **Q** for groups of size *m* up to 32, with *k* up to 10. These values should suffice in practice. Next, we show similar but 32 x 20 portions of **P'** and **Q'** for $GF(2^8)$ used in the examples. The entries of **P'** are now $GF(2^8)$ elements given as two hexadecimal digits. For change, the entries of **Q'** are given in decimal as logarithms numbers between 0 and 254. The program to generate the complete matrices should be requested from the authors at [CERIA].

0001	0001	0001	0001	0001	0001	0001	0001	0001	0001
0001	eb9b	2284	9e44	f91c	7ab9	2897	41f6	a9dd	5933
0001	2284	9e74	d7f1	0fe3	79bb	5658	efa6	30f3	641c
0001	9e44	d7f1	75ee	512d	4e14	16bb	2ce0	36c8	0f9a
0001	f91c	0fe3	512d	59c3	d037	b205	cb3c	f6e2	c606
0001	7ab9	79bb	4e14	d037	b259	e9b9	2c40	81b2	70b5
0001	2897	5658	16bb	b205	e9b9	c7b6	07c7	8670	86ac
0001	41f6	efa6	2ce0	cb3c	2c40	07c7	2c2c	5ddc	148f
0001	a9dd	30f3	36c8	f6e2	81b2	8670	5ddc	7702	1f19
0001	5933	641c	0f9a	c606	70b5	86ac	148f	1f19	9c98
0001	52d5	59c3	94f7	4d4d	e9b2	40f1	2d00	9c04	bc3f
0001	3f68	3d2b	00f1	32c2	dfb2	6ab4	c6a6	9eba	0241
0001	47b5	cb3c	6f1f	2d00	39e6	799c	c83b	92df	c24d
0001	8656	b46f	59c3	903a	7432	9aef	46f5	3b50	e867
0001	db71	b612	eb07	496e	ac26	74c2	04cc	5f5d	23b9
0001	92fe	acb3	3045	fef2	7607	ad10	2df0	0b2f	1eaa
0001	99f1	6d93	5803	1ce8	4099	136c	af32	35b6	3274
0001	2c68	2d00	4fef	50bf	16af	88ec	2ec0	bfa2	2b90
0001	7502	c6a6	8f58	5a03	2887	89ca	6724	e0be	39e1
0001	227d	16a8	eacf	0f59	67af	7702	838d	3517	85a1
0001	1027	496e	a06a	c486	61ab	131f	2e00	b405	fafc
0001	0a46	87b4	74c2	f85b	e8ef	5b03	3163	8b11	b4a5
0001	27df	9523	379a	7e8d	0301	0221	7702	fe93	d06b
0001	3dad	27bc	9f64	7602	5cf1	eef2	2e48	64a0	a751
0001	8dd7	f6e2	f125	9c04	f720	c0a3	92df	ee07	1d73
0001	1e27	fdd3	93fd	86cd	9ca4	2614	9961	8483	c26e
0001	0cf7	46f5	2d00	cf2f	6f7b	2f80	8497	4baf	2900
0001	651e	a0d6	58d3	dbb0	96ed	f57a	2f20	5c03	69d3
0001	71ec	0f8d	496e	7702	51a4	f85b	ba6f	a34b	ce4b
0001	00a1	fda5	564f	dc72	6924	b699	9d44	de6d	a90c
0001	b4ca	b385	025b	0eb8	47bd	31f6	f173	a768	798b
0001	59c3	73b6	b325	4b4b	6ba7	7ca5	2fd0	5e55	9ac4

Figure 14 Generic parity matrix **P'** for $GF(2^{16})$: first 32 rows by 10 columns.

0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
0000	5ab5	e267	784d	9444	c670	9df5	bcbf	05b6	54ff
0000	e267	0dce	2b66	0e6b	181c	ff62	f5b1	08c1	cf6b
0000	784d	2b66	a3b3	c9f8	f273	b13f	d9ba	f2b9	7b58
0000	9444	0e6b	c9f8	050d	70c8	f6a1	3b14	ff05	a7a4
0000	c670	181c	f273	70c8	3739	44c5	1f1d	1916	bfc0
0000	9df5	ff62	b13f	f6a1	44c5	f604	580f	2baf	9e26
0000	bcbf	f5b1	d9ba	3b14	1f1d	580f	14cf	05e7	e2b8
0000	05b6	08c1	f2b9	ff05	1916	2baf	05e7	06e7	0131
0000	54ff	cf6b	7b58	a7a4	bfc0	9e26	e2b8	0131	5630
0000	cc60	050d	285b	3159	1542	f5a7	0607	fe25	dad7
0000	8e54	3e62	b0f9	dd25	2b69	d79b	e606	142d	86c4
0000	ae54	3b14	d0d9	0607	7778	005d	2ec8	07e1	966f
0000	ac60	04ed	050d	5d68	1e23	1e69	0ec8	edd0	ae8b
0000	782d	d2b9	9ea6	d3b3	0bd0	ca55	3216	db37	7377
0000	e287	08e1	d2d9	27e7	21f7	57de	0ee8	0100	b809
0000	a753	27c7	caf2	3333	209a	f87b	ea2e	8460	5851
0000	b361	0607	f8c0	0d8b	5250	e0d0	0de8	b6b3	320c
0000	8593	e606	aaf2	df1f	da83	0acf	ee84	5b9f	6a2e
0000	d521	07c7	d8c0	e934	1ffd	06e7	41fb	ea50	2b0b
0000	81ab	d3b3	80f4	d1d9	e4e8	eb35	d2d3	4f53	81bb
0000	d909	e1b0	ca55	f781	169f	ef1d	264a	05c6	22a7
0000	8fa8	3a1a	d95d	25ed	173c	fd88	06e7	dc12	2c05
0000	cb0c	2c1d	22bf	1bd8	5cb5	1534	2aa0	a212	a169
0000	d9a6	ff05	30e3	fe25	a1ff	502e	07e1	8286	2424
0000	810e	39a4	9dac	5c9d	eaef	cc2f	3347	5033	fdde
0000	1445	0ec8	0607	5f93	20f7	fe82	98ab	ab47	2b25
0000	4670	d429	72d0	a0f6	11bf	225c	a6fd	1c96	ec01
0000	381e	6073	d3b3	06e7	4c49	f781	6dd1	b793	3389
0000	2297	7dfd	a0a2	a86f	9539	d3a8	7c23	0121	d474
0000	55a8	ad5a	0311	a579	257a	a5d6	e187	2ad4	b4c6
0000	050d	8fd0	cfff	6416	1642	29d5	0cee	64d4	2a2b

Figure 15 Generic logarithmic parity matrix **Q'** for $GF(2^{16})$: first 32 rows by 10 columns.

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1a	1c	a0	cd	7d	b1	e5	30	48	2c	26	68	52	f4	3	de	19	4e	45
1	3b	37	a9	d4	7c	f9	84	4f	5b	93	63	5	f6	a7	d3	89	9f	31	a2
1	ff	fd	2c	cc	30	48	85	76	68	52	da	3	7	ba	f4	d0	cd	7d	29
1	16	b6	fb	6d	4c	bf	75	50	b8	9	f2	38	c5	9d	b5	57	cb	3d	6c
1	90	4b	52	18	76	68	1e	d1	3	7	b	f4	71	98	ba	77	cc	30	32
1	86	14	93	e0	4f	5b	e4	d	5	f6	6a	d3	ef	aa	a7	23	d4	7c	ae
1	87	5f	68	52	41	b0	40	f0	f	3	a5	c2	f4	96	e3	88	2c	12	e7
1	94	24	74	a3	6e	11	fa	d9	9b	60	e9	c6	2f	72	e8	59	af	80	d5
1	3a	59	3	7	f0	f	13	cf	c2	f4	c1	e3	ba	58	96	73	52	41	3d
1	2e	e1	f6	15	d	5	bc	3f	d3	ef	33	a7	95	3e	aa	e7	e0	4f	a
1	4d	c7	a8	f8	17	d5	56	ce	9e	67	f4	25	43	66	42	74	5a	d6	44
1	d7	19	9	4a	50	b8	c0	2b	38	c5	12	b5	d6	de	9d	82	6d	4c	ca
1	a1	fc	1b	61	3	ed	9f	f4	57	a2	21	82	ae	80	7f	64	2e	68	8b
1	fe	54	5	f6	9a	d8	32	10	6f	d3	bd	81	a7	83	7e	df	93	c7	7f
1	1f	6c	7	b7	d1	3	91	df	f4	71	af	ba	20	2a	98	31	18	76	6b
1	70	35	85	44	e	f1	a8	b1	40	1e	b4	13	91	ed	f3	ac	28	c3	cc
1	84	ca	71	b2	df	f4	fe	9c	ba	20	a3	98	bf	51	2a	7c	b7	d1	62
1	1b	ac	d3	ef	10	6f	6b	79	81	a7	e5	7e	aa	56	83	9c	f6	9a	80
1	f9	c	b4	4c	e0	d7	f4	15	c4	39	ac	8	b9	8a	fa	eb	3d	d4	81
1	c4	cd	c5	fc	2b	38	a4	99	b5	d6	41	9d	17	d0	de	7f	4a	50	34
1	89	99	c2	f4	e4	eb	7c	bc	35	e3	11	cb	96	4a	6d	86	3	84	b4
1	7d	39	a1	b0	3a	cc	88	45	18	dc	a7	b7	eb	8b	b2	85	49	87	cf
1	dc	1d	a2	75	f4	57	d4	ba	82	ae	70	7f	a	6e	80	f9	61	3	fb
1	ee	9f	60	1a	d9	9b	8a	5c	c6	2f	c7	e8	87	89	72	97	a3	6e	9e
1	e2	f4	22	d0	b6	28	5c	19	44	88	6b	3b	73	2e	86	92	de	b3	49
1	55	ff	4b	8	d3	5f	cb	a7	59	6c	65	97	ca	c3	e2	20	c4	5	c1
1	45	97	f4	71	cf	c2	f3	16	e3	ba	74	96	98	5e	58	e9	7	f0	4c
1	fd	a0	38	c5	b9	51	a	be	4d	b5	1f	3c	9d	22	e6	ab	9	39	e2
1	3c	9a	67	65	ce	9e	ec	f1	25	43	ba	42	24	a9	66	60	f8	17	3b
1	61	d5	ef	36	3f	d3	1c	ab	a7	95	28	aa	78	c	3e	3d	15	d	db
1	f4	e6	79	ab	8b	c0	d8	fb	a4	94	e	37	ee	e1	14	e0	3f	b2	5e

Figure 16: Generic parity matrix **P'** for $GF(2^8)$: first 32 rows by 20 columns.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	105	200	55	12	243	86	169	29	226	240	15	107	148	230	25	62	193	34	221
0	120	185	135	41	115	214	140	136	92	119	163	50	173	205	82	74	46	181	209
0	175	80	240	127	29	226	128	121	107	148	134	25	198	57	230	108	12	243	147
0	239	93	234	133	16	162	21	54	132	223	213	201	123	32	42	189	236	228	250
0	227	179	148	28	121	107	76	161	25	198	238	230	94	17	57	43	127	29	194
0	99	52	119	203	136	92	156	104	50	173	40	82	215	151	205	47	41	115	190
0	13	64	107	148	191	242	6	79	75	25	188	67	230	180	176	103	240	224	81
0	38	225	10	91	186	100	244	96	217	30	245	164	69	155	11	210	97	7	157
0	9	210	25	198	79	75	14	246	67	230	45	176	57	241	180	159	148	191	228
0	130	89	173	141	104	50	71	166	82	215	125	205	184	114	151	81	203	136	51
0	145	118	144	116	129	157	219	111	137	110	230	36	98	126	139	10	19	85	102
0	170	193	223	37	54	132	31	218	201	123	224	42	85	62	32	192	133	16	73
0	63	168	248	66	25	117	46	230	189	209	138	192	190	7	87	195	130	107	237
0	88	143	50	173	146	251	194	4	61	82	109	112	205	247	167	90	119	118	87
0	113	250	198	158	161	25	165	90	230	94	97	57	5	142	17	181	28	121	84
0	202	39	128	102	199	174	144	86	6	76	20	14	165	117	233	220	53	216	127
0	140	73	94	211	90	230	88	35	57	5	91	17	162	208	142	115	158	161	182
0	248	220	82	215	4	61	84	212	112	205	169	167	151	219	247	35	173	146	7
0	214	27	20	16	203	170	230	141	183	154	220	3	60	222	244	235	228	41	112
0	183	12	123	168	218	201	149	68	42	85	191	32	129	108	62	87	37	54	106
0	74	68	67	230	156	235	115	71	39	176	100	236	180	37	133	99	25	140	20
0	243	154	63	242	9	127	103	221	28	187	205	158	235	237	211	128	152	13	246
0	187	8	209	21	230	189	41	57	192	190	202	87	51	186	7	214	66	25	234
0	44	46	30	105	96	217	222	131	164	69	118	11	13	74	155	124	91	186	137
0	95	230	101	108	93	53	131	193	102	103	84	120	159	130	99	153	62	171	152
0	150	175	179	3	82	64	236	205	210	250	72	124	73	216	95	5	183	50	45
0	221	124	230	94	246	67	233	239	176	57	10	180	17	70	241	245	198	79	16
0	80	55	201	123	60	208	51	65	145	42	113	77	32	101	160	178	223	154	95
0	77	146	110	72	111	137	122	174	36	98	57	139	225	135	126	30	116	129	120
0	66	157	215	249	166	82	200	178	205	184	53	151	78	27	114	228	141	104	177
0	230	160	212	178	237	31	251	234	149	38	199	185	44	89	52	203	166	211	70

Figure 17: Generic logarithmic parity matrix **Q'** for $GF(2^8)$: first 32 rows by 20 columns.

APPENDIX B DEFINITION OF TERMS

Term	Description	Typical value
	Addressing	
F	an LH*$_{RS}$ file	
i	file level	initially 0, scales monotonically
n	split pointer	$0 — 2^i - 1$
(i, n)	file state	
$(\tilde{\imath}, \tilde{n})$	internal file state in a data bucket	
i'	client's view of i	$0 — i$
n'	client's view of n	$0 — n$
N	current number of data buckets in the file	$2^i + n$
a	logical address of a data bucket server	[0; 1; 2;…;M - 1]
A	physical address of a data bucket server	IP address
A_0	initial physical address of the file (server of data bucket 0)	IP address
j	data bucket level	i or $i + 1$
c	(primary) key of a data record	random; $0 — 2^{32} - 1$
h_i	series of hash functions	C mod N * 2^i
α	load factor	0.6 — 1.0
	Parity calculus	
$GF(2^f)$	Galois Field of size (2^f)	$GF(2^{16})$
F	Galois Field of size (2^8)	
ECC	Erasure Correcting Code	
RS	Reed-Solomon Code	
B	parity field (in parity record)	$GF(2^{16})$ symbols
C	record group structure field (in parity record)	$c_0, c_1, \ldots c_{m-1}$
k	bucket (record) group local availability level	1 — 10
K_{file}	global file availability level	1 — 10
g	bucket group number	1,2…
r	data record rank (and parity record key)	1 — b
α	primitive element in GF	$\alpha = 2$
$\log_\alpha(\xi)$	logarithm of symbol ξ ; $\xi \in GF(2^f)$, $\xi \neq 0$	Table 1
antilog (i)	antilogarithm of integer i ; $0 \leq i < 2^f - 1$	Table 1
I	identity matrix m x m	
P'	generic parity matrix	Figure 14
P	actual parity matrix	upper left m x K submatrix of **P'**
Q'	generic logarithmic parity matrix	Figure 15
Q	actual logarithmic parity matrix	upper left m x K submatrix of **Q'**
H, H^{-1}	decoding matrices (m x m, formed from avail. columns of **P**)	
L_A	list of available buckets in a bucket group recovery	m
L_S	list of spare buckets for a bucket group recovery	$l \leq k$

File Param.	Description	Typical value
B	bucket capacity (records per data bucket)	50 — 1,000,000
K	intended file availability level (scales monotonically)	1 — 5
m	bucket group size (also max. record group size)	4 — 32