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

WITOLD LITWIN,
RIM MOUSSA
U. Paris Dauphine
and
THOMAS SCHWARZ, S.J.,
Santa Clara University[1]

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 $k \geq 1$ of its 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 Reed-Salomon erasure correcting coding. The resulting parity storage overhead is about the lowest 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 gained in

[1] This research was supported partially by a generous gift from Microsoft Research, Europe and EEC-ICONS project no. IST-2001-32429.
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, TSchwarz@calprov.org.

popularity with the emergence of grid computing and P2P applications. They need new data structures that scale well with the number of components [CACM97]. Scalable Distributed Data Structures (SDDS) aim to fulfill this need [LNS93], [SDDS]. An SDDS file is stored at multiple nodes provided by *SDDS servers.* As the file grows, so does the number of servers on which it resides. The SDDS addressing scheme has no centralized components. This allows for operation speeds independent of the file size. They provide for hash, range or m-d partitioned files of records identified by a primary or by 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 use [CERIA].

Among the best-known SDDS schemes is the LH* scheme [LNS93, LNS96, KLR96, BVW96, B99a, K98v3, R98]. LH* creates scalable, distributed, hash-partitioned files. Each server stores the records in a bucket. The buckets split when the file grows. The splits follow the *linear hashing* (LH) principles [L80a, L80b]. Buckets are stored for fast access in distributed RAM, otherwise they can be on disks. Only the maximum possible number of server nodes 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].

At times, an LH* server can become unavailable. It may fail as the result of a software or hardware failure. It may also stop the service for good or an unacceptably long time, a frequent case in P2P applications. Either way, access to data becomes impossible. The situation may not be acceptable for an application, limiting the utility of the LH* scheme. Data unavailability can be very costly, [CRP06]. An unavailable financial database may easily cost the owner $10K-$27K per minute, [B99].

A file might suffer from the unavailability of several of its servers. We say that it is *k-available*, if all data remain available despite the unavailability of any *k* servers. The information-theoretical minimum storage overhead for *k*-availability of *m* data servers is *k*/*m* [H&al94]. It requires *k* additional, so-called parity symbols (records, buckets…) per *m* data symbols (records, buckets…). Decoding *k* unavailable symbols requires access to *m* available symbols of the total of $m + k$. Large values for *m* seem impractical. A reasonable approach to limit *m* is to partition a data file into groups consisting of at most *m* nodes (buckets) per group, with independent parity calculus.

For files on a few servers, 1-availability usually suffices. The parity calculus can be then the fastest known, using only XORing, as in RAID-5. The probability of a server unavailability increases however with the file size. The file *reliability*, which is the probability that all the data are available for the application, declines necessarily. At one point we need the 2-availability despite the increased storage overhead. Likewise, as the

file continues to grow, at some point we need 3-availability, despite further increase to the storage overhead. In fact, for any fixed *k*, the probability of *k*-unavailability increases with the file size. For largely scaling files, we need the *scalable availability*, adjusting *k* to the file size [LMR98].

Below, we present an efficient scalable availability scheme we called LH*$_{RS}$. It structures the LH* data buckets into groups of size *m*, providing each with $K \geq 1$ parity buckets. The values of *m* and of *K* are file parameters that can be adjusted dynamically. We call *K* the *intended availability level.* A group is typically *K*-available. Some can be (*K*-1)-available with respect to the current *K* if the file just increased it by one. The new level diffuses to the groups progressively with the splits. Changes to *K* value are transparent for the application.

The scheme's theory originates in [LS00]. Since then, we have sped up the parity calculus, without compromising the storage overhead. We also have progressively optimized implementation issues, especially the efficiency of the communication architecture. We have realized a prototype for the multi-computer of Wintel PCs on a typical LAN [LMS04]. We have analytically and experimentally determined various performance factors. The work has validated our design choices. Below we report on all these issues.

Our current parity calculus is a novel erasure correction scheme, using Reed-Solomon (RS) coding. Our storage overhead for *k*-availability is close to the optimal one of *k*/*m*. To our best knowledge, our schemes offer the fastest parity generation (encoding) for our needs. We have also optimized the erasure correction (decoding), although with a lesser priority, expecting it much less frequent, hopefully.

More specifically, we recall that an RS code uses a Galois Field (GF). The addition in a GF amounts to XORing only. Multiplication is necessarily slower [MS97]. In departure from [LS00], we determined the $GF(2^{16})$ and $GF(2^{8})$ more appropriate. Our scheme uses now only XORing to generate the first parity symbol (record, bucket, …). We also only need XORing for the recovery of a single data bucket. In addition, changes to a data record in the first bucket of any group result in XORing only at the parity buckets, further speeding up 1/*m* of our encoding operations. Moreover, we have accelerated the *k*-parity encoding and decoding, by introducing the *logarithmic* parity and decoding matrices. All this makes our parity encoding scheme the fastest known, under the minimal group storage overhead constraint, to our best knowledge of the *k*-erasure correction codes. Besides, our high-availability features are transparent to the application

and do not affect the speed of searches and scans in LH*$_{RS}$ file. These perform as well as in an LH* file with the same data records and bucket size.

LH*$_{RS}$ is the only high-availability SDDS scheme prototyped to the extend we present. However, it is not the only one known. Some proposals use mirroring to achieve 1-availability [LN96, BV98, VBW98]. Two schemes use only XORing to provide 1-availability [L&al97, LR01, L97]. Another XORing-only scheme LH*$_{SA}$ was the first to offer scalable availability [LMRS99]. It can generate parity faster than LH*$_{RS}$, but has sometimes greater storage overhead. We compare various schemes in the related work section.

To address the overall utility of the LH*$_{RS}$ scheme, we recall that the existing hash file schemes store files on local disk(s), or statically partition (parallelize) them over a cluster of nodes. The latter approach provides for larger files or better efficiency of the (non-key) scans. As we have mentioned, the disks may use underneath a hardware or software RAID scheme, 1-available RAID5 typically. This one may adversely affect a record access performance, e.g., by segmenting some records, or manipulating blocs much larger than records, as it works independently of the hash scheme. Partitioned files are sometimes replicated, for 1-availability typically given the incurred storage overhead cost.

The LH*$_{RS}$ scheme is "plug compatible" with such schemes. It offers the same functional capabilities: the key search, a scan, a record insert, update and delete. The access performance should in contrast typically improve by orders of magnitude. Especially for files that can now entirely fit for processing into potentially unlimited (distributed) RAM of a multicomputer, while they could not at a single node or a cluster. Our experiments with LH*$_{RS}$ files in distributed RAM showed the individual key search about 30 faster than a single access to a local disk. A bulk search speeds up 200 times. Experiments with inserts lead to similar figures, for up to 3-available file. The application may accelerate thus the query processing from, - let's say - 30 minutes or 3 hours to a single minute.

Likewise, as we said, the administrator may choose the availability level "on-demand" or may let it to autonomously adjust to the file size. The storage overhead is always about the minimal possible. Both properties are uniquely attractive at present for larger and very large (Pbyte) files. Records are never segmented and data transfers from the storage are not bigger, as the high-availability is designed to work in accordance with the hash scheme in this way. Our experiments showed furthermore that the recovery from a *k*-unavailability should be fast. For instance, about 1.5 sec sufficed to recover more

than 10MB of data (100 000 records) within three unavailable data buckets. Next, the file may become very large, spreading dynamically over theoretically any number of nodes. The scaling transparency frees finally the application administrator from the periodical file restructuring. Some already eagerly wait for the perspective to materialize, [BM03].

Hash files are ubiquitous. Legions of their applications could benefit from the new capabilities of LH*$_{RS}$. This is particularly true for numerous applications of major DBMSs. DB2 and Oracle use single site or parallel hash files, and Postgres DBMS uses the (single site) linear hash files. Other applications affected by the current technology include video servers, Web servers, high performance file servers, or dedicated mission-critical servers. In the latter category, the rapidly and constantly growing data of a well-known large-scale search engine may allegedly already need about 54.000 nodes, [E03].

The unrivaled at present scalability of LH*$_{RS}$ files should also serve the emerging needs of data grids and of P2P applications, as we discuss more in Section 6. In particular, a number of new applications target there files larger than anything operationally workable till now, e.g., the SkyServer project to cite just one, [G02]. On a different tune, the sheer access speed to data in an LH*$_{RS}$ file together with its ability to quickly scale, makes it an attractive tool for stream analysis, In this increasingly popular domain, streams often come simultaneously from multiple sensors, needing the storage at the generation speed. Incidentally, the problem of collection of such streams from an airplane engine, at Santa Clara University mechanical lab, triggered the original LH* scheme idea.

Below, we describe the general structure of an LH*$_{RS}$ file in Section 2. Section 3 presents the parity calculus. We explain the LH*$_{RS}$ file manipulations in Section 4. Section 5 deals with the performance analysis. Section 6 discusses the related work. Section 7 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. We give additional details in Appendix C (on-line) regarding the parity calculus, file operations, performance analysis and variants of the basic scheme, including the discussion of alternative erasure correcting codes.

2 FILE STRUCTURE

LH*$_{RS}$ provides high availability to the LH* scheme [LNS93, LNS96, KLR96, LMRS99]. LH* itself is the scalable distributed generalization of Linear Hashing (LH) [L80a, L80b]. An LH*$_{RS}$ file stores *data* records in data buckets, numbered 0,1,2… and *parity* records in separate parity buckets. Data records contain the application data. A data bucket has the *capacity* of $b >> 1$ *primary* records. Additional records become

overflow records. The application interacts with the data records as in an LH* file. Parity records only provide the high availability and are invisible to the application.

We store the data and parity buckets at the LH*$_{RS}$ *server* nodes. There is basically one bucket per server. The application does not address any server directly. It calls an LH*$_{RS}$ *client* component, usually at the application node. The clients address the servers for data search or storage over the network, transparently for the applications.

An LH*$_{RS}$ operation is in *normal* mode as long as it does not encounter an unavailable bucket. If it does, then it enters *degraded* mode. Below, we assume the 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

The storage and addressing of data records in an LH*$_{RS}$ file is the same as in an LH*$_{LH}$ file, [KLR96]. We thus only briefly recall the related principles. Details are in [KLR96], as well as in [LNS96], [BDNL00], [B02]. We follow up with the description of the LH*$_{RS}$ specific rules for *misdirected* requests.

2.1.1 *Storage*

A data record structure consists of a *key* identifying it and of other, non-key, data. Figure 2a, shows the record structure, with *c* denoting the key and *D* the non-key field, usually much longer than the key field. The key determines the record location (the bucket number *a*) through the LH-function [L80a]:

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

Here, h_i stands for a family of hash functions with specific properties. Usually, one uses $h_i = c \bmod 2^i$. The variables (i, n) are the *file state*, where $i = 0, 1\ldots$ stands for the *file level* and $n = 0,1\ldots$ is the *split pointer*. The state, hence the LH-function itself, dynamically adjusts to the file size. The state determines the number N of data buckets in the file as $N = 2^i+n$. We call N *file extent*. Vice versa, the extent determines the file state. For every N, 2^i is the largest power of 2 smaller or equal to N.

The initial file state is (0,0) for the file extent $N=1$. Every record hashes then to bucket 0. LH* file adjusts to the scale-up by bucket splits. A low ratio of overflows result from typically and good access performance. If the file shrinks, buckets can merge, to prevent an under-load. It appears that merges are not used in practice for file systems. Therefore, we forego further discussion.

An LH*$_{RS}$ component called the *coordinator* manages the splits (and merges). It may reside anywhere, but it should be practical to locate it at the node of bucket 0[2]. Both components remain then however distinct, i.e., the unavailability of bucket 0 does not imply that of the coordinator and vice versa. The coordinator can be *k*-replicated for its own *k*-availability, having very little data on its own, as it will appear. When an insert causes an overflow, the bucket informs the coordinator. The coordinator sends the split request to bucket *n*. It also appends to the file bucket $N = 2^i+n$, allotted at some server, according to some allocation scheme, [LNS96]. Bucket *n* readdress (rehashes) every record in it using h_{i+1}. A crucial property of LH-functions is that h_{i+1} only maps each key to *n* or to 2^i+n. The records mapped to *n*, half of those in the bucket on the average, remain. The others move to new bucket. Both buckets retain the value $i + 1$ as the *bucket level*. The initial level of bucket 0 is zero.

Each split increases *n* to $n + 1$. The resulting (LH) calculus with the new state conveniently takes the new bucket into account. The buckets split in this way in numerical (linear) order, until all the buckets in the file reach the level $i + 1$. This occurs when all the buckets 0 to $2^i–1$ have split using h_{i+1}. The coordinator increases then the file level to $i := i + 1$ and moves the split pointer back to $n := 0$. The splitting process uses from now on the new h_{i+1}.

The internal structure of an LH*$_{RS}$ data bucket is that of an LH*$_{LH}$ bucket. Each bucket is basically a RAM LH file, as in [L80a]. The internal buckets called pages are the chains of records, the overflow records are those beyond the *b*-th element. The result provides for efficient page splitting and fast local access. Perhaps more surprisingly, it also accelerates the LH*$_{RS}$ splits, with respect to their naïve processing.

Example 1

Consider the LH*$_{RS}$ data buckets at Figure 1a. The split pointer is $n = 4$, the file level is $i = 3$, the file extent is $N = 12 = 2^3 + 4$ buckets. The header of each bucket shows its level *j*. We only show the key fields *c*, assumed to be integers. The bucket capacity is $b = 4$. Bucket 4 has an overflow record. Its insertions triggered one of the previous splits. Most likely, - one of those using h_4 that processed the file up to bucket 3 till now. We now insert record 77. According to (LH) it enters bucket 5. The bucket reports an overflow to the coordinator. This one creates bucket $N =12$, Figure 1b, and requests bucket 4 to split. Bucket 4 has level $j = 3$, hence rehashes all its records using h_4. The new address for a record can be only either 4 or 12. The bucket keeps every record hashed to 4. It sends

[2] For LH*, there are also variants without the coordinator, [LNS96], but we are not aware of attempts to make them highly-available.

every other record to bucket 12. In our case, two records remain, while three move. Both buckets get level $j = 4$. Bucket 4 reports the end of the split to the coordinator that finally moves the split pointer n to $n + 1 = 5$. Now, bucket 4 no longer overflows, but even has room for three more primary records. Likewise, there is room for two more inserts in new bucket 12. The overflow at bucket 5 remains though, despite being at the origin of the split. Its resolution waits for coming splits. In our case, the next one is in fact at bucket 5 and will indeed resolve the overflow.

2.1.2 *Addressing*

As in LH* file, each LH*$_{RS}$ client caches its private image of the file state, and applies (LH) to it. The coordinator would indeed become a hot spot if the clients should access it for the addressing. The coordinator does not push every file state update to the clients nor even to the servers for the same reason. A split makes therefore every existing image outdated. The initial image of a new client is (0,0). A client with an outdated image may direct a key-based request towards an *incorrect* address, i.e., not the *correct* one given by (LH) for the request. The addressee recognizes its status from the received key c and from its bucket level j. The correct address must indeed be equal to h_j (c) that is its own.

The incorrectly addressed bucket should forward the request to the bucket that is possibly the correct one for c, and within the shortest extent N that could exist in the situation. This guarantees that the hop does exceed any current extent. The guess can be based solely on j and the bucket's own address. Indeed, only the coordinator knows n. The first guess must be bucket $h_{j-1}(c)$. This address must be under 2^{j-1}, hence the guess is always safe, as the extent of the file with at least one bucket of level j must be beyond that bound. If this address is not that of the guessing bucket itself, then it cannot verify whether the level of the addressee is actually $j - 1$ or j. Hence, it has to resend the request there. Otherwise, the guess does not lead to any actual forwarding. The only remaining choice is $h_j(c)$. This must be the correct address. If the request is resent to bucket $h_{j-1}(c)$, and the address happens to be also incorrect, then the bucket level was indeed equal to j. The same calculus there resends then the request to the correct bucket $h_j(c)$.

An incorrect bucket receiving the request from the client resends it accordingly to the above reasoning. It executes the resulting LH*$_{RS}$ *Forwarding Algorithm* shown in Section 8 in Appendix C. The addressee may resend it once more, using the same algorithm. This must be the last hop. In other words, any client's request dealt with in this way must reach its correct bucket in at most two hops. This property is independent of the file extent. It is unique to LH* based schemes up to now, and comparatively, among SDDSs, the most efficient known at present. Besides, the performance analysis for LH*

has shown that most of the key based addressing requests in fact do not encounter any forwarding in practice, i.e., for a reasonable large bucket capacity b[3].

The correct bucket sends an *Image Adjustment Message* (IAM) to the client. It contains essentially the bucket level and the address of the latest resending bucket[4]. The client updates its image on this basis. It guesses the shortest file extent and the related state that could lead to the data it got. The algorithm avoids the same addressing error twice. It also makes the client's image of the extent closer to the actual one. Numerically, see Section 8 of Appendix C, if j is the received level, and A is the address, the image evaluates to ($i' = j - 1$, $n' = A + 1$), unless n' turns out then to $2^{i'}$. The file state reevaluates then instead to ($i' = j$, $n' = 0$). The rationale for the calculus is largely similar to that already outlined for the forwarding calculus at the bucket.

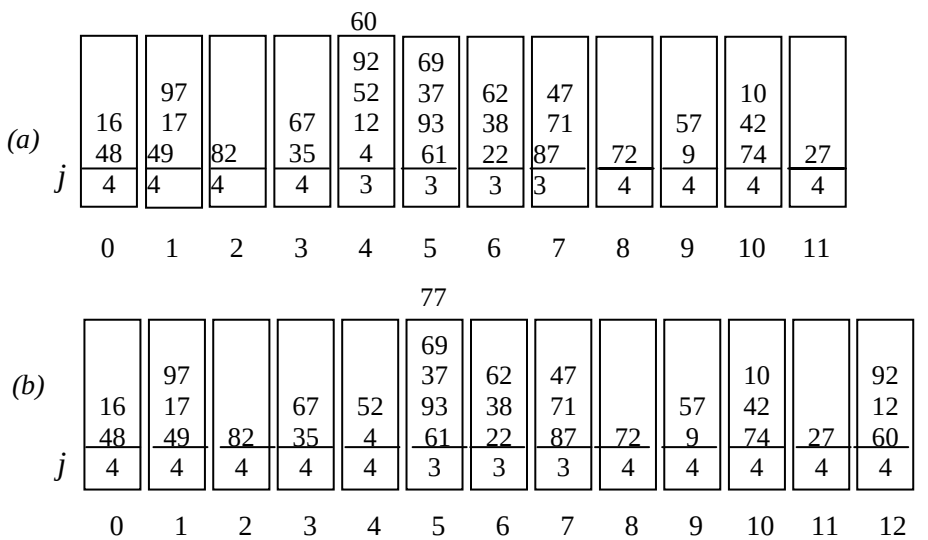

Figure 1: An LH*$_{RS}$ file data buckets, before (a) and after (b) the insert of record 77

The IAMs may also refresh the client's image of the physical addresses of the servers. Each IAM brings to the client then the addresses of all the buckets stretching between the one initially addressed by the client and the correct one. A specific IAM we describe later refreshes also the client when the recovery displaces a bucket.

Example 2

In the file at Figure 1a, the key $c = 60$ is in bucket 4 according to (LH). Now, a client with the file state image equal to (0,0), hence with the file extent image equal to $N' = 1$,

[3] Incidentally, these properties nicely fit D. Knuth's comments about amazing efficiency of randomized algorithms in Lecture 2 of "Things a Computer Scientist Rarely Talks About", CSLI Publications, Stanford, 2001.

[4] This choice is actually a novelty for LH* based schemes. They used up to now the one in [LNS96]. The new approach improves the image adequacy in many cases.

searches for this key. The client sends the request to bucket 0, according to its execution of (LH). Bucket 0 verifies whether it could be the correct bucket for record 60. Its bucket level is 4, hence we have $h_4(60) \neq 0$. The bucket is not the correct one, hence needs to forward the request. The bucket guesses the new address as $h_3(60) = 4$. It is safely in the existing file extent, as the file with bucket 0 of level 4 must have at least 9 buckets, while h_3 may only address buckets 0…7. The forwarding address does not point to bucket 0 itself, hence the bucket forwards the request to bucket 4. It includes in the message its own bucket level $j = 4$. The calculus at bucket 4 yields $h_3(60) = 4$. Hence it is the correct bucket. It responds therefore to the client with record 60 (if the request was a key search). It piggy-backs the IAM containing the address 0 and the received level $j = 4$. The client updates the file state image to (3,1). This amounts to a new extent image of $N' = 9$. This value is much closer to the actual one of $N = 12$ than the initial $N' = 1$. If the client repeats the search, it sends the request directly to bucket 4, avoiding the previous error. Moreover, initially any request from the client for any but the two records in bucket 0 would require the forwarding. Now, only the requests for the six records in buckets 9-11 would require it.

Consider now another client with image of $N = 1$ sending the request for record 60 to the file at Figure 1b. Bucket 0 again forwards the request, and its bucket level, to bucket 4. The calculus at bucket 4 now yields $h_4(60) = 12$. The request cannot be resent to bucket $h_3(60)$ since it is bucket 4 itself. Hence the bucket uses the guess of $h_4(60)$ and resends the request accordingly, with its own bucket level to bucket 12. This has to be the correct address. Bucket 12 handles the request and sends the IAM that the level of bucket 4 is 4. The client's new image is (3,5) hence it amount to the image $N' = 13$. It is the perfect guess. An access to any record by our client arrives now directly at the correct bucket, until next split.

2.1.3 *Misdirected Requests*

LH*$_{RS}$ restores an unavailable bucket on a spare server. It is typically a different server than the one with the unavailable bucket. Only the client and the buckets involved in the recovery are aware of the new location. Any other client or bucket can *misdirect* a request to the former server. A misdirected request could also follow a bucket merge. The LH*$_{RS}$ file processes the misdirected requests basically as follows.

Any request to a data bucket carries the number of its intended bucket. A node that a request reaches is supposed basically to be an SDDS server. Other nodes are not supposed to react in any way. Like, *ipso facto*, an unavailable server. The request without reply enters the degraded mode that we discuss in Section 4. The server getting a request

verifies that it carries the intended bucket. If so, it acts as described above. Otherwise, it forwards the request to the coordinator. This one resends the request to the correct bucket. An IAM informs the sender of the location change.

The scheme is valid provided that a misdirected request never finds the bucket at former location available despite its recovery elsewhere. If it could happen, the client could unknowingly get not up-to-date data. The assumptions for LH*$_{RS}$ design we outline now prevent this eventuality. They root in the popular fail-stop model and fit well the local network environment we target mainly.

We presume thus basically that when a client or server finds a bucket unavailable, the unavailability is permanent and of the entire bucket, Next, we suppose the communications (fully) reliable. Finally, we presume the server (definitively) unavailable when it does not react to the requests from the coordinator. The basic recovery policy for LH*$_{RS}$ is furthermore that (i) only the coordinator initiates a recovery, when a bucket does not react to the request from, and that (ii) the first action of any server upon its local start-up or restart is to report to the coordinator. The bucket waits then for the reply before providing any services, even if it finds its bucket intact.

Reliable communications and (i) prohibit then any server from the recovery while available. This excludes one possibility of the “bad” case. Next, the server restarting on its own could have its bucket already recovered by the coordinator, in the meantime, or not, if its unavailability remained unspotted. The coordinator systematically instructs any restarting server with the recovered bucket to become a spare. This precludes the bad case as well. If the unavailability remained unspotted, and the bucket is intact, the coordinator allows the bucket to serve. Otherwise, finally it initiates the recovery. The restarting server may eventually get the bucket back. Whatever is the issue, the bad case cannot happen.

The scheme can be extended beyond the basic model. Especially, to the case of unreliable communications that could temporarily or partly isolate a data bucket. The bucket could then become inaccessible to a client and the coordinator, while other clients could possibly continue accessing it. The coordinator would start the recovery that if successful, could ultimately lead the clients using the former location to get stale data. To prevent this from happening, it suffices that any data server pings (scrubs) any of the parity buckets of its group at the interval shorter than a minimal bucket recovery time. In practice, as Section 5.7 shows, it should lead to a negligible overhead, e.g., a ping per little bit more than second in our experiments. If the server does not receive the reply, it alerts the coordinator, while probing possibly another parity bucket.

As it will appear, every parity bucket knows the valid location of all the data buckets in its group, or knows about any recovery of a data bucket in its group in progress. In response to the ping, the available parity bucket should thus either confirm to the sender that it is still the available data bucket, or make it aware of the recovery in progress. Accordingly, the bucket should process any request coming afterwards, or during the ping, or should become a spare, only forwarding the misdirected requests to the coordinator. Notice that if the request is an insert, delete or update, it must update all k parity buckets, as it will appear. These would redirect any such request to the valid bucket, if a, necessarily unfinished, recovery had started since the latest ping. The misdirected key search, processed in contrast by the (invalid) data bucket alone, as any key search by any data bucket, cannot lead to an incorrect (stale) reply in this case either. No update could indeed perform at the valid bucket yet, at least till next ping.

2.2 Parity Records

The LH*$_{RS}$ parity records let the application to get the values of data records stored on any unavailable servers, up to $k \geq 1$. We call k the *availability level* and the file *k-available*. The actual level depends on the *intended availability level* $K \geq 1$ that is a file parameter. The K value adjusts dynamically to the file size, (Section 2.2.3). Typically, $k = K$, sometimes $k = K–1$, after an increase of K. We now present how LH*$_{RS}$ manages the parity records.

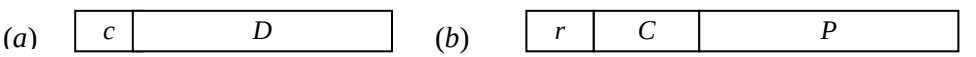

Figure 2: LH*RS record structure: (a) data record, (b) parity record

2.2.1 *Record Grouping*

LH*$_{RS}$ parity records belong to the specific structure, invisible to the application. It consists of *bucket groups* and *record groups*. A bucket group contains m consecutive buckets with possibly less buckets in the last group. Formally, bucket group g consists of all buckets a such that $\lfloor a/m \rfloor = g$. Here, m is a file parameter that is a power of 2. Bucket group 0 consists of buckets $0…m–1$, bucket group 1 of buckets $m…\ 2m–1$, etc. We limited our implementation to $m \leq 128$, since larger choices look of little use at present. Every data record in a bucket gets a unique *rank* $r = 1,2….$ when it enters the bucket because of an insert, split, or merge. Ranks are handed out basically successively, although the ranks of deleted records can get reused. The up to m data records sharing the same rank r in a bucket group g form a *record group* (g,r). Each record group has $k \geq 1$ parity records stored respectively at different *parity bucket* $P_0, ... P_{k-1}$. The value of k is

the same for every record group in a bucket group and follows that of K as we have mentioned and describe in depth in Section 2.2.3.

2.2.2 *Record Structure*

Figure 2b shows the structure of a parity record. Field r contains the rank and serves as the key. Field C encodes the record group structure. It contains m placeholders c_0, $c_1 \ldots c_{m-1}$ for the keys of the data records in the group. If the i^{th}-bucket in the group contains a data record with rank r and key c, then $c_i = c$ otherwise c_i is null. All k parity records in a record group share the value of r and of C. The final field is the *parity* field P, different for each parity record of a group. We generate it by encoding the D-fields in the record group using our Erasure Correcting Code (ECC) (Section 3). The ECC allows to decode any unavailable (hence assumed erased) $s \leq k$ D-fields in the group from any of $s \leq k$ parity records and the remaining $m–s$ D-fields. We can recover the unavailable s keys from the C-field of any parity record in the group. These properties are our basis for the k-availability.

In [LS00], the actual keys for C formed a variable length list. The fixed structure above, proved more efficient, [Lj00]. It typically needs slightly less storage. In addition, the position i of c in the C-field directly identifies the data bucket with c as the i^{th} in its bucket group. This facilitates the search for data record c, that needed the LH* addressing, with the image management at the parity bucket and possibly the forwarding.

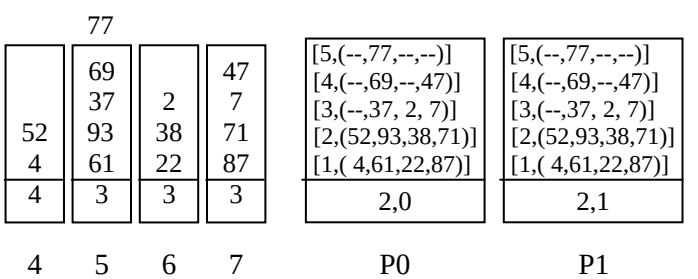

Figure 3: LH*$_{RS}$ Record group structure with k = 2 parity buckets

Example 3

We continue with our running example at Figure 1. We assume now that the bucket group size is m = 4. The file at Figure 1b has then four bucket groups {0,1,2,3}, {4,5,6,7}, {8,9,10,11}, and {12}. Figure 3 shows the 2nd group that is group 1, with $k=2$ parity buckets. These are named P0 and P1. We only show the keys in the data buckets, the ranks and the C-fields in the parity buckets. Notice that the latter are identical for

both buckets, but that the *P*-fields would differ. The hyphens symbolize the null values. The header of a parity bucket only shows its bucket group number and the offset of the bucket in the group. The figure shows that the record group with rank 1 consists of *m* data records 4, 61, 22, and 87. The group of rank 3 contains only three records at present, in buckets 5,6,7. The next insert to bucket 3 will add a new member to this group, and will update all the *C* and *P* fields. If bucket 6 is unavailable, the data records in buckets 4,5,7 together with any of the parity buckets suffice to correct the erasure of the *D* fields in this bucket. The missing keys are in the *C* fields at the offset $i = 3$. To access record 93, among those necessary for the erasure correction calculus within its record group (1,2), one may directly address bucket 5. Since $93 = c_1$ in its *C*-field, we calculate 1*4 +1. A 2-unavailability involving buckets 6,7, requires the use of data buckets 4,5 and of both parity buckets. A 3-unavailability is unrecoverable (catastrophic) here. We need at least $k = 3$ parity buckets to keep away from such a bad luck.

2.2.3 *Scalable Availability*

The cost of storing and manipulating parity records increases with *k*. The storage overhead is at least *k/m*. We also need the access to all the *k* parity records whenever we insert, update, or delete a data record. A multiple unavailability becomes also more likely in a larger file, hence the probability of the catastrophic case for any given *k* rises as well, [H&a94]. In response, the LH*$_{RS}$ scheme provides *scalable availability*, [LMR98]. At certain file sizes, the current availability levels increase by one for every group.

In more detail, we maintain a file parameter called the *intended availability level K*. Initially, $K = 1$. The coordinator increases *K* by 1, at the first split making the file extent *N* to exceed some N_K data buckets, $K \geq 2$. Each N_K should be a power of 2 and a multiple of *m*. The former requirement makes *K* changing only for the split of bucket 0. One choice for the latter requirement is $N_K = N_1^K$, with $N_1 = m$, [LMR98].

Consider now that the coordinator is at the point of increasing *K* since the file undergoes the split of bucket 0 making *N* exceeding some N_K. At this moment, all bucket groups yet contain $k = K–1$ parity buckets. Now, whenever the first bucket in a bucket group splits, its group gets one more parity bucket, i.e., the *K*-th one. From now on, every new split within the group updates this bucket as well. If the freshly appended bucket initiates a new group, then it gets *K* parity buckets from the start. Until the file reaches $N = 2N_K$ buckets, we have thus in the file some groups which are *K*-available while others are still (*K*-1)-available. The file is still (*K*-1)-available. Only when *N* reaches $2N_K$ will all the groups have *K* buckets, which makes the (entire) file *K*-available.

There is a subtle consequence for the *transitional* group that contains the current bucket *n* beyond its first bucket. Not all the buckets of the transitional group have yet split using h_{i+1}. Its *K*-th parity bucket only encodes therefore the data buckets in the group up to bucket *n*-1. LH*$_{RS}$ recovery cannot use it in conjunction with the data buckets in the group that have not yet split. The group remains $(K - 1)$-available despite *K* parity buckets. It reaches *K*-availability when the last bucket splits.

Example 4

We continue with $m = 4$. The file creation with $K = 1$ leads to data bucket 0 and one parity bucket for group 0, i.e., with buckets 0…3 as $m = 4$. The split creating bucket 4 initialize the first parity bucket for group 1, named P0 upon Figure 3 and Figure 5 and named P_0 formally. Until the file reaches $N = 16$, every group has one parity bucket (P0), and the file is 1-available. The creation of bucket 16, by the split of bucket 0, appends P1 to group 0, and provides group 4 initiated by bucket 16, with P0 and P1 from the start. Group 0 remains transitional and 1-available, despite its two parity buckets. This lasts until bucket 3 splits. The new group 4 is however 2-available immediately.

Next, the eventual split of bucket 4 appends P2 to group 1, and starts group 5. Group 1 is transitional in turn. When bucket 15 splits, hence *n* returns to 0 and the file reaches $N = 32$ buckets, all the groups are 2-available. They remain so until the file reaches $N_3 = 64$ buckets. Next split, of bucket 0 necessarily, increases *K* to 3, adds P3 to group 0, makes the group transitional again, etc. The file reaches 3-availability when it attains 128 data buckets. And so on.

3 PARITY CALCULUS

3.1 Overview

We first defined the parity calculus for LH*$_{RS}$ in [LS00] for our Erasure Correcting Code (ECC) derived from a classical Reed-Solomon code [MS97]. We recall that an ECC encodes a vector **a** of *m* data symbols into a *code word* of $m + k + k'$ code symbols such that any $m + k'$ of the code symbols suffice to recalculate **a**. Our ECC was Maximum Distance Separable (MDS). Hence $k' = 0$ and *m* code symbols suffice to recalculate **a**. This is the theoretical minimum, so we minimize parity storage overhead for any availability level *k* within a record group. Next, our ECC was systematic. Thus, our encoding concatenates a vector **b** of *k* parity symbols to **a** to form the code word (**a**|**b**). Finally, our code was linear. Thus, we calculate **b** as $\mathbf{b} = \mathbf{a} \cdot \mathbf{P}$ with a *parity matrix* **P**.

Our initial proposal was theoretical. Since then, we worked on the implementation. Our primary goal was fast processing of changes to data buckets. The result was an improved ECC that we introduce now. We changed from symbols in the Galois field

GF(2^4) to GF(2^8) or GF(2^{16}). We based our latest prototype, [LMS04], on the latter as measurements favored it. We also refined the parity matrix **P**. Our new **P** has a row and a column of ones. As a consequence of the latter, updating parity bucket P_0 involves only bit-wise XOR operations. In addition, the recovery of a single data bucket involves only the XOR operations as well. Our EEC calculus is thus in this case as simple and fast as that of the popular 1-available RAID schemes. The 1-unavailability is likely to be the most frequent case for our prospective applications. When we need to go beyond, the row of ones makes the inserting, deleting, or updating the first data record in a record group, to involve only XOR-operations at all parity records. Creation and maintenance of all other parity records involves symbol-wise GF-multiplication. It is mathematically impossible to find a parity matrix for an MDS code with more ones in it, so that our code is optimal in that regard. Our new parity matrix is new for ECC theory, to the best of our knowledge.

We continue to implement our GF-multiplication using logarithms and antilogarithms, [MS97, Ch. 3, §4]. We take further advantage of this approach by directly applying to the parity (*P*-fields*)* generation the logarithms of the entries in **P,** instead of the entries themselves. The latter is the usual practice and as we did so ourselves as well previously. The result speeds up the calculus, as we will show. Finally, we introduce the concept of a *generic parity matrix*. This matrix contains parity matrices **P** for a variety of numbers *m* of data records and *k* of parity records.

We now describe our present calculus in more depth. We only sketch aspects that our earlier publications already covered. See in particular [LMS04b] for more details. We give some details also in Appendix C.

3.2 Generic Parity Matrix

Our symbols are the 2^f bit strings of length *f*, *f* = 8, 16. We treat them as elements of GF(2^f). The generic parity matrix $\mathbf{P}_{gen}$ is the 2^{f-1} by $2^{f-1}+1$ matrix of the form:

$$\mathbf{P}_{gen} = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & p_{1,1} & \cdots & p_{1,2^{f-1}} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & p_{2^{f-1}-1,1} & \cdots & p_{2^{f-1}-1,2^{f-1}} \end{pmatrix} \tag{3.1}$$

With this choice of size $\mathbf{P}_{gen}$ can accommodate a maximum of 2^{f-1} data records and $2^{f-1}+1$ parity records in a group. We could have chosen other maxima within the bounds of a total of 2^f+1 data and parity records in a group. We now sketch how to obtain $\mathbf{P}_{gen}$. We start with an extended Vandermonde matrix **V** with $r = 2^{f-1}$ rows and $n = 2^f+1$

columns over GF(2^f). This is a matrix of largest known size so that any r by r sub-matrix is invertible. Elementary row transformations and multiplication of a column by a scalar preserve this property. We use them to transform **V** into a matrix **W** of the form (**I**|*), i.e. a matrix whose left half is an identity matrix. We now multiply all rows j of **W** with $w_{r,j}^{-1}$. Column r now only contains ones. However, the left r columns are no longer the identity matrix. Hence we multiply all columns $j \in \{0,\ldots r\text{-}1\}$ with $w_{r,j}$ to recoup the identity matrix in these columns. Next, we multiply all columns r,..., 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 $\mathbf{G}_{\text{gen}}$. Matrix $\mathbf{P}_{\text{gen}}$ is its right half.

$$\mathbf{G}_{\text{gen}} = \begin{pmatrix} 1 & 0 & \cdots & 0 & 1 & 1 & \cdots & 1 \\ 0 & 1 & \cdots & 0 & 1 & p_{1,1} & \cdots & p_{1,r} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 1 & p_{r-1,1} & \cdots & p_{r-1,r} \end{pmatrix}. \tag{3.2}$$

Appendix A shows our actual generic parity matrices for GF (2^8) and GF (2^{16}). The Lemma in Section 10 in Appendix C shows that any parity matrix obtained as the upper left corner of our generic parity matrix defines a systematic, linear, MDS ECC.

3.3 Parity Generation

We recall that the addition in a GF with 2^f elements is the bit-wise XOR operation. Also, among several methods for the multiplication, the popular method of logarithms and antilogarithms [MS97] is especially convenient for our purpose. Given a primitive element $\alpha \in GF(2^f)$, we multiply two non-zero GF elements β and γ as $\beta \cdot \gamma = \text{antilog}_\alpha(\log_\alpha(\beta) + \log_\alpha(\gamma))$. We tabulate the logarithms and antilogarithms. There are 2^f-1 entries in the logarithm table (one for each non-zero element) and twice as many in the antilogarithm table, one for each possible sum. By adding the logarithms modulo 2^f-1, we could use a smaller antilogarithm table at the cost of speed. See Section 9 in Appendix C for more details of our use of GF arithmetic, including the pseudo-codes. Table 5 there shows the logarithms based on $\alpha = 2$ we actually use for GF (2^8).

We organize the non-key fields of the m data records in a record group as the columns in an l by m matrix $\mathbf{A} = (a_{i,j})$, $0 \le i < l$, $0 \le j < m$. Similarly, we number the parity records in the group from 0 to $k-1$ and arrange their P-fields as the columns in an l by k matrix $\mathbf{B} = (b_{i,j})$, $0 \le i < l$, $0 \le j < k$. Parity matrix **P** is the upper left m by k corner of $\mathbf{P}_{\text{gen}}$. We define **B** by $\mathbf{B} = \mathbf{A} \cdot \mathbf{P}$. Equivalently, we have:

(3.3) $$b_{i,j} = \bigoplus_{\nu=0}^{m-1} a_{i,\nu}\, p_{\nu,j}\,.$$

Formula (3.3) defines the *P*-field of the parity records for a record group. We calculate this field operationally, when we change a data record in the record group. This happens when an application inserts, deletes, or modifies a data record, or when a split or merge occurs. We implemented the latter as bulk inserts and deletes. For the parity calculus, inserts and deletions are special cases of updates. A missing data record has a non-key field consisting of zeroes. An insert changes this field from the zero string and a delete changes it to the zero string.

We now explain an update to data record *j* in a record group. Let $\left(a_{i,j}^{\text{old}}\right)_{0\le i<l}$ be the non-key field of the record before and $\left(a_{i,j}^{\text{new}}\right)_{0\le i<l}$ the one after the update. The *delta field* (Δ-field) is $\mathbf{\Delta}^{j} = \left(\delta_{i,j}\right)_{0\le i<l} = \left(a_{i,j}^{\text{new}} \oplus a_{i,j}^{\text{old}}\right)_{0\le i<l}$. It is the bit-wise XOR of the before and the after image of the non-key field. For an insert or a delete, the Δ-field is thus simply the actual non-key field. As in [LS00], we calculate the P-field of parity record *s* as:

(3.4) $$b_{i,s}^{\text{new}} = b_{i,s}^{\text{old}} \oplus \delta_{i,s} \cdot p_{j,s}\,.$$

Parity bucket *s* only uses column *s* of **P**. However, we do not use the parity matrix **P** directly. Rather, we store the logarithms $q_{j,s} = \log_{\alpha}(p_{j,s})$ of the entries of **P** in a logarithmic parity matrix **Q** and use the formula

(3.5) $$b_{i,s}^{\text{new}} = b_{i,s}^{\text{old}} \oplus \text{antilog}_{\alpha}(\log_{\alpha}(\delta_{i,s}) + q_{j,s})$$

where the plus sign is the integer addition and the ⊕ the bitwise XOR operation. Our use of **Q** avoids half the look-ups to the logarithm table. Parity bucket *s* only needs the first *m* coefficients from column *s* of **Q** and only these are stored there. Parity bucket 0 does not need to store anything, since it simply updates according to

(3.6) $$b_{i,s}^{\text{new}} = b_{i,s}^{\text{old}} \oplus \delta_{i,s}\,.$$

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

Figure 4: **P** and **Q** for 3-available record group of size four data record.

Appendix A shows our actual matrices **Q** for GF (2^8) and GF (2^{16}). At the implementation level, a data bucket calculates the Δ-field for each update and sends it together with the rank to each parity bucket. Each parity bucket then updates the *P*-field

of the parity record in the record group given by the rank according to (3.5) or to (3.6) if it is the first parity bucket.

Example 5

We use 1B symbols as elements of $GF(2^8)$ and write them as hexadecimal numbers. We continue with m = 4 and we choose k = 3. We cut the parity matrix **P** and a logarithmic parity matrix **Q** in Figure 4 from our generic parity matrix in Appendix A. We number the data buckets in a bucket group D0…D3 and consider three parity buckets, Figure 5. We have one data record per bucket. The record in D0 has the *D*-field: "En arche en o logos …". The other *D*-fields are "In the beginning was the word …" in D1, "Au commencement était le mot …" in D2, and "Am Anfang war das Wort…" in D3. Assuming the ASCII coding, *D*-fields translate to (hex) *GF* symbols in Figure 5c, e.g., "45 6e 20 61 72 63 68 …" for the record in D0. We obtain the parity symbols in P0 from the vector $\mathbf{a}^0$ = (45,49,41,41) multiplied by **P.** The result $\mathbf{b}^0 = \mathbf{a}^0 \cdot \mathbf{P}$ is (c, d2, d0). We calculate the first symbol of $\mathbf{b}^0$ simply as 45 ⊕ 49 ⊕ 41 ⊕ 41 = c. This is the conventional parity, as in a RAID. The second symbol of $\mathbf{b}^0$ is 45·1 ⊕ 49·1a ⊕ 41·3b ⊕ 41·ff etc. Notice that we need no multiplication to calculate the first addend.

(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	44	9	f6	fe
6e	6e	6d	61	c	54	9
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

(d)

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 5: Example of Parity Updating Calculus.

We now insert these records one by one. Figure 5a shows the result of inserting the first record into D0. The record is in fact replicated at all parity buckets, since updates to the first bucket translate to XOR operations at the parity buckets, and there are not yet the other data records. Inserting the second record into D1 at Figure 5b still leads to an XOR operation only at P0, but involves GF-multiplications using the respective **P** columns at the other parity buckets. Notice that, in the latter case, we operationally use in fact **Q** columns only. Figure 5c shows the insert of the last two records. Finally, Figure 5d

shows the result of changing the first record to “In the beginning was …”. Operationally, we first calculate the Δ-field at D_0: (45⊕49, 6e⊕6e, 20⊕20, 61⊕74, …) = (c, 0, 0, 15, …) and then forward it to all the parity buckets. Since this is an update to the first data bucket, we update the P-fields of *all* parity records by only XORing the Δ-field to the current contents.

3.4 Erasure Correction

Our ECC calculates a code word (**a**|**b**) as (**a**|**b**) = **a**·**G** with a *generator matrix* **G** = (**I**|**P**) [MS97, Ch. 1, §2] obtained by the concatenation of the *m* by *m* identity matrix **I** and the parity matrix **P**. We recall that we organized the non-key fields of the data records in an *l* by *m* matrix **A** and similarly the P-fields of the parity records in a matrix **B = A·P**. Assume that we have *m* columns of (**A**|**B**) left, corresponding to *m* surviving records. We collect the corresponding columns of **G** in an *m* by *m* sub-matrix **H**. Here, the data buckets implicitly correspond to columns in the identity matrix. The columns of **P** are reconstructed as the coordinate-wise antilogarithms of the columns in **Q** at the parity buckets. We form an *l* by *m* matrix **S** containing the *m* available data and parity records. We have **A**·**H** = **S**. Hence $\mathbf{A} = \mathbf{S}\cdot\mathbf{H}^{-1}$ and we recover all data records with one single matrix inversion (in our experiments, the classical Gaussian inversion appeared to be the fastest, and we actually use $\log_2(\mathbf{H}^{-1})$ to speed up the multiplication). If there are unavailable parity records, then we recover the data records first and then regenerate the erased *P*-fields by calculating **B** = **A**·**P** for the lacking columns of **B**.

Alternatively, we can decode all the erased data and parity values 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 $\mathbf{(A|B)} = \mathbf{A}\cdot\mathbf{G} = \mathbf{S}\cdot\mathbf{H}^{-1}\cdot\mathbf{G} = \mathbf{S}\cdot\mathbf{R}$. We apply however only the first approach up to now. Computing **R** is longer, while we expect primarily the recovery of the data bucket only.

$$H = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 1 & 1a & 1c \\ 0 & 1 & 3b & 37 \\ 1 & 1 & ff & fd \end{pmatrix} \qquad H^{-1} = \begin{pmatrix} 1 & a7 & a7 & 1 \\ 46 & 7a & 3d & 0 \\ 91 & c8 & 59 & 0 \\ d6 & b2 & 64 & 0 \end{pmatrix}$$

Figure 6: Matrices for correcting erasure of buckets D0, D1, D2

Example 6

Assume that the data buckets D0, D1, and D2 in Figure 5 are unavailable. Assume that we want to read our record in D0. We collect the columns of **G** corresponding to the available buckets D3, P0, P1, and P2 in Figure 5 in matrix **H**, Figure 6. We invert **H**. The last column of $\mathbf{H}^{-1}$ is a unit vector since the fourth data record is among the available ones. To reconstruct the first symbol in each data bucket simultaneously, we form a

vector **s** from the first symbols in the available records of D3, P0, P1, and P2: **s** = (44, 5, fa, f2). This vector is the first row of **S**. To recover the first symbol in D0, we multiply **s** by the first column of $\mathbf{H}^{-1}$ and obtain 49 = 1*44 + 46*5 + 91*FA + d6*f2. Notice again that actually we use the matrix $\log_2(\mathbf{H}^{-1})$. We iterate over the other rows of **S** to obtain the other symbols in D0. If we were to read our record in D1, we would use **S** with the second column of $\mathbf{H}^{-1}$.

4 LH*$_{RS}$ FILE MANIPULATION

An application manipulates an LH*$_{RS}$ file as an LH* file. The coordinator manages high-availability invisibly to the application. We assume that the coordinator can diagnose any bucket unavailability by *probing* through reliable communications. Available buckets respond to the probing as expected, by definition. Internally, each manipulation starts in normal mode. It remains so as long as it does not run into an unavailable bucket. That one does not respond at all (our basic case) or more generally does not behave as it should. A manipulation that encounters an unavailable bucket enters *degraded mode*. The node handling the manipulation forwards it to the coordinator. The coordinator initiates the *bucket recovery*, unless it is already in progress. It considers the bucket content erased and calls upon our erasure correction calculus. The recovery restores simultaneously *all* unavailable buckets found in the group when the operation starts, up to *k*, as we discussed. More than *k* unavailable buckets make the unavailability catastrophic.

For a key search of a record in an unavailable bucket, the coordinator starts also the *record recovery*. This operation recovers only the record, or finds its key not in the file. It should be usually much faster than waiting for the bucket recovery to complete, which also recovers the record. Once the bucket recovery, or only the record recovery for the key search, has terminated, the coordinator completes the original manipulation and returns the result and the control to the requestor.

We first present the recovery operations. Next, we describe the record insert, update and bucket split operations. We focus on the parity coding, as Section 2 already largely covered the related data record manipulations. We have left the description of file creation and removal, of key search, and of non-key search (scan) to Appendix C, Section 11. These manipulations do not involve parity coding and perform almost as for LH*, except for the degraded mode. Likewise, we relegated the deletion and bucket merge to Appendix C. Deletions are rare (except those internal to the split operation we describe in Section 4.5), while merges are hardly ever implemented.

4.1 Bucket Recovery

The coordinator starts the bucket recovery by probing the *m* data and *k* parity buckets of

the bucket group for availability. We usually have $k = K$, or we have $k = K - 1$ if the group is not aware yet of the current K or is the transitional one.. The probing should localize m available buckets, excluding the last parity bucket in a transitional group. The un availability is catastrophic otherwise. The coordinator reports the gloomy fact to the client, for the application, and stops the processing. It may still be possible to recover some records, but this is beyond the basic scheme.

The probing localizes also the unavailable buckets that should be therefore k at most, including the originally reported bucket and without the last bucket of the transitional group[5]. If $k > 1$, then prior to the erasure correction, the coordinator may need to verify whether all the available parity buckets processed the last Δ-record or even simply got it. It may not be the case, as we show in Section 4.3 and Section 4.4. If Δ-record could update some but not all parity buckets, the erasure correction result could be gibberish, corrupting the data. The coordinator terminates the update of the parity buckets in the need. Next, the coordinator establishes the list L_A of m available buckets. L_A possibly includes bucket P_0. The coordinator establishes also the list L_S of tuples: (unavailable bucket, replacing spare). The coordinator chooses one spare as the *recovery manager*. It passes the task to it. If there is any parity bucket in L_S, it also passes its **Q** column. The handover prevents the coordinator from becoming a hot spot.

The manager first creates the empty, structure of the bucket to recover at each spare. Next, if there is only one unavailable (data) bucket, and $P_0 \in L_A$, then the manager applies the XOR-only erasure correction. Otherwise, it creates matrix **H** using the columns of **Q** at the parity buckets in L_A. If the bucket group is the last one and some data buckets have not yet been created, then the calculus considers these buckets as virtual ones with records having zero D-fields. In both cases, the manager calculates then the matrix $\log_2(\mathbf{H}^{-1})$, using the primitive element $\alpha = 2$ and the Gaussian inversion to produce $\mathbf{H}^{-1}$, as we said in Section 3.4. Next, it starts looping over all the parity records in a parity bucket in L_A. It requests the successive records and for each record received, it performs the *record group* recovery producing all the unavailable records of one group.

Record group recovery explores the *C*-fields, Figure 2a. For every key $c_i \in C$, it requests the data record c_i from the i^{th} bucket in the group, provided it is in L_A. The manager decodes the (erased) *D*-fields of unavailable data records in the group. It uses the XOR-only or $\log_2(\mathbf{H}^{-1})$. It also reconstructs the keys from *C*. If there are also

[5] In the basic scheme, we exclude the case of the probing finding a bucket reported to be unavailable to be nevertheless available, because for example only the reporting client had problems communicating with the bucket.

unavailable parity buckets in L_S, then the manager generates their records from the m data records. The procedure varies slightly for the last parity bucket in a transitional group. The recovery of this bucket uses only the data buckets among the m up to bucket n (non-included). Finally, the manager sends the recovered records to the spares.

Once the bucket group recovery ends, the manager sends the addresses of the recovered buckets to the remaining buckets in the group. These update the location tables. The manager finally returns the control to the coordinator. The coordinator updates its server addresses as well. Clients and servers get the new addresses when they misdirect the requests.

It is perhaps worth recalling furthermore that the erasure correction used for each record group recovery could reconstruct the data and parity buckets simultaneously, except for the last bucket of the transitional group. It could use the alternate erasure correction calculus discussed in Section 3.4. The unavailability involving only the data buckets may be expected however more frequent than that of both types together. Our current erasure correction is then faster.

While the bucket group recovery loops over the individual record recovery, the actual records sent among the data buckets, the parity buckets and the spare move in bulks. At present, we use the TCP/IP in passive mode, Section 5.1. This turns out to be by far more effective than our previous choice of UDP with a record per datagram.

4.2 Record Recovery

A record recovery results from a key search in an unavailable correct bucket a. It decodes only the requested record or finds that the key is not in the file. This suffices for completing the search and should be typically much faster than delaying the key search till the sufficient completion of the bucket recovery, (Section 5.7 and Section 5.8). The coordinator starts the record recovery in parallel to a bucket recovery, provided the unavailability is recoverable of course. It hands the control to the *record recovery manager* at an available parity bucket in the group. It transmits c, a, L_A. The manager first searches the C-fields in the bucket for the existence of a parity record with rank r and the searched key c in $C(r)$. Since one cannot infer r from c, it basically scans the bucket. It visits in each record only the C-field value at the offset of the unavailable bucket in the group. An index at the parity bucket, giving the ranks of the keys in the C-fields, could obviously avoid the scan, at the additional storage and processing cost. Such index is not part of the basic scheme.

If the manager does not find the parity record, it informs the coordinator that the search is unsuccessful. Otherwise, the recovery manager uses the erasure correction on

the record group *r* to recover the searched *D*-field. As discussed already, the calculus uses possibly XOR only or $\mathbf{H}^{-1}$ otherwise. Unlike for the record group recovery within the bucket recovery, the record recovery manager restores only the requested record, even if there are more unavailable data records in the group. Finally, it sends the record to the coordinator that forwards it to the client.

4.3 Insert

In normal mode, an LH*$_{RS}$ client performs an insert like an LH* client. The client addresses then the insert to the bucket determined by the data record key *c* and the client's image. It keeps the copy of the record and waits for an acknowledgement. If it does not come within a timeout, the client sends the insert to the coordinator. The operation enters the degraded mode. We have already presented the bucket recovery that it starts. Later in this section we address the aspects of the degraded mode specific to the insert. The client waits for the final acknowledgment to discard its copy of the record. The policy for its acknowledgements to the application is implementation dependent and is not within our basic scheme. Our prototype uses its flow control algorithm.

The bucket receiving the request eventually forwards it as described before. Once 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-field is the *D*-field of the inserted record. The data bucket then commits (acknowledges) the insert to the client and waits, internally, for the *k* acknowledgements from all parity buckets. The client discards its copy of the record.

The policy provides the *k*-availability under reliable messaging and our other basic assumptions, provided that all the available parity buckets perform the required update, as detailed below. The data bucket will report indeed any unavailability and will still have the Δ-record. If the probing during the recovery finds any $l \leq k$ parity buckets available, all *l* will be updated with the Δ-record. If the probing shows also at most *l* data bucket unavailable, then our basic erasure correction calculus recovers first all of them and next the unavailable parity buckets. This, even if the probing even finds the original data bucket among the unavailable ones, as it could become unavailable in the meantime.

To update its content upon the message from the data bucket, each parity bucket creates the parity record *r*, if it does not exist already, and inserts *c* into *C*-field. It also encodes the Δ-field of Δ-record *c* through the update of *P*-field of record *r*. It first conceptually multiplies the Δ-field symbol-wise with the correct coefficient of matrix **P**.

The actual calculus does not use the multiplication for the first column as it has 1's only, and uses the symbols in **Q** for $k > 1$. It either XORs the result to the *P*-field already in record *r* or stores it as the new *P*-field of new record *r*.

If $k = 1$, the send-out of the Δ-record amounts in our scheme to 1PC. That is, the parity bucket creates or updates the parity record *r*, acknowledges the operation and does not keep any other traces of it. The data bucket erases any traces as well after the acknowledgement to the client. That one does the same, perhaps after forwarding the acknowledgement to the application in turn. The server or the client enters the degraded mode when any of the expected messages does not arrive in time.

The approach does not suffice in contrast to provide *k*-availability for $k > 1$. For instance, assume that $k = 2$ and that both the data bucket and the client become unavailable, while the data bucket was sending the messages to the parity buckets. The unavailability of the data bucket can then be discovered only later on. It may happen then that the bucket could send the message only to one of the parity buckets that performed the creation/update of record *r* consequently. Consider that the probing finds yet another data bucket unavailable. The erasure correction calculus obviously cannot recover the group anymore, leading to the data corruption in the case of such an attempt. This includes the inserted record itself that, besides, is nowhere around anymore. In any case, the group, hence the file, is not *k*-available anymore, despite the *k* parity buckets in the group. Notice that for an update the case could occur even if only the data bucket became unavailable during the send-out. Unlike for an insert or delete, the *C*-field of a parity bucket does not suffice anymore to know whether it dealt with the Δ-record or not.

The general rule for $k > 1$ that follows from is that a change, whether insert, update or delete, should commit at all or none parity buckets. Our basic scheme uses the following simple variant of 2PC for any of these operations. The data bucket sends the Δ-record *c* and its *r* to all *k* parity buckets. It does it in the order of the column index in **P**, i.e., p_0, $p_1 \ldots p_k$. Each parity bucket starts the commit process by acknowledging the reception of the message. The confirmation constitutes the "ready-to-commit" message. 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 sends out the "commit" message to the *k* buckets, in the same order. Each bucket that receives the message discards the Δ-record. All available buckets should receive it under our general assumptions.

Degraded mode starts when any of the buckets involved cannot get a response it expects. The operation at the data bucket enters degraded mode if it lacks any of the

acknowledgments from the parity buckets. It alerts the coordinator, transmitting the Δ-record, *r* and as usual the number *p* of the unavailable parity bucket. Within the bucket recovery process as described above, the coordinator requests each available bucket to complete the update from the differential file or by using the Δ-record it sends out otherwise. The recovery process proceeds then further as already presented.

Another degraded case occurs when parity bucket p_l does not receive a message from the data bucket. The data bucket must then just have failed and bucket p_l must be in the "Ready-to-Commit" state and must still have the Δ-record. It alerts the coordinator, sending out the Δ-record, and *r*. The coordinator probes the parity buckets in the above order. Any bucket up to p_l either has committed or still has the Δ-record. Any other bucket either still has the Δ-record or did not get it at all. The coordinator sends the Δ-record where needed and requests the update at all the available parity buckets accordingly. Again, the recovery process continues then as already presented.

Next, the client lacking the acknowledgement from the data bucket reports it as unavailable. The client also acknowledges the operation to the application, to avoid further waiting. The coordinator performs the delivery to the correct bucket if the unavailable one wasn't. This may generate a split processed then as usual. The coordinator may also find that the correct bucket it found on its own is unavailable as well. This may trigger a separate recovery if the bucket is in a different bucket group than the one reported by the client. Next, the coordinator determines the availability of buckets in the group, and verifies the synchronization of the parity buckets as just described. It may find that the data bucket never sent any messages to parity buckets. Alternatively, it may find an alert also from a parity bucket about the data bucket's unavailability. It may further find some parity buckets still in wait for the commit from the data bucket and some perhaps without any message from. In all cases, the coordinator can determine the state of each available parity bucket and synchronize all of them as above. Afterwards the recovery proceeds as described.

Finally, as we said, it may happen that the client and the data bucket become simultaneously unavailable during the insert. If the update was in progress, an available parity bucket would alert the coordinator as described. Alternatively, it may only be that no (available) parity bucket has received the Δ-record or all have nicely committed. Only a later application operation will discover the data bucket unavailability. It will be recovered accordingly with or without the inserted record.

An insert in a degraded mode to the unavailable correct data bucket may generate an overflow at the recovered bucket. The new bucket itself alerts the coordinator to perform a split. On the other hand, observe that one can optimize the 2PC to use multicast messaging to the parity buckets for inserts and deletes. It is due to the possibility of the recovery synchronization using the *C*-fields.

Finally, notice that, while our 2 PC is a sure solution for *k*-availability and 1 PC cannot be, the event of a data bucket becoming unavailable during the messaging to the parity buckets is very unlikely. Section 5.4 shows this time to be under a millisecond. The bad case of both the client and the data bucket becoming unavailable during that time is obviously even much more remote. An application may be reasonably tempted to use 1PC anyhow, for its simplicity and necessarily better performance. After all, every application already disregards scores of potential, but fortunately very unlikely, causes of data unavailability.

4.4 Update

An update operation of record *c* changes its non-key field. In the normal mode, the client performs the update as in LH*. As for an insert, it waits for an acknowledgement. The client sends the received record with its key *c* and the new value of the *D*-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 $k = 1$. To be on the safe side for $k > 1$, the updates also basically use the 2PC above. It clearly suffices for this operation as well.

4.5 Split

As in LH*, if an insert to an LH*$_{RS}$ data bucket *a* results in an overflow, then the bucket *a* notifies the coordinator. The coordinator starts the *split* operation of bucket *n*, determined by the split pointer. We recall that this operation is invisible to the application. 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 a 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* empty parity buckets. If $K > k$ for the bucket group with bucket *n*, then the coordinator allocates the additional K^{th} parity bucket. Provided all this performs normally, the coordinator sends the *split* message to bucket *n* with all the addresses. This hands control of the split to bucket *n*. The coordinator waits

nevertheless for the commit message. The bucket sends all the data records that change address when rehashed using h_{j+1} to data bucket *N*. 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. Each existing bucket, deletes Δ-record from parity record *r* and inserts into parity record *r'*. The new K^{th} parity bucket, if there is one, disregards the deletes.

When data bucket *N* receives the data records, it requests the insert into its *K* parity buckets with the successive ranks it assigns. Once it terminates, it reports to bucket *n* that in turn reports to the coordinator.

The operations on the parity buckets use 1PC for *K* = 1 and the already presented 2PC otherwise. Degraded mode starts when a data or a parity bucket does not reply. We skip the discussion of this mode here, as various cases are similar to those already discussed. Once the split terminates, the coordinator adjusts the file state as in Section 2.1.1.

5 PERFORMANCE ANALYSIS

We have analyzed storage, communication, and processing performance of the scheme. We have derived formulae for the load factor, parity storage overhead, and messaging costs. We limited the derivation to the dominant cost factors. This makes the calculus easy enough, but still quite lengthy. This analysis is in Appendix C (Section 12).

The result show that the high availability of an LH*$_{RS}$ file incurs the storage overhead about the smallest possible. More precisely, for any intended availability level *K*, and of group size *m*, the load factor of the growing LH*$_{RS}$ file should be in practice about constant and the highest possible for these values, as well as for any technique added to an LH* file to make it *K*-available. Through motivating examples in Appendix C, we show some practical design choices.

A user is mainly interested in the response time of various operations. The complexity of any practical implementation of LH*$_{RS}$ seems to prevent a practically useful formal analysis of such times. We have preferred the experimental analysis of various implementations. We now present the results. They complete the picture of the efficiency of our scheme and they validate the various design choices that we presented above.

5.1 Prototyping LH*$_{RS}$

We have implemented LH*$_{RS}$ to measure various operations and prove the viability of the scheme. The work took many years of effort. The earliest prototype is presented in [Lj00]. It implemented the parity calculus defined in [LS00]. It also reused an LH*$_{LH}$ implementation for the data bucket management, [B02]. Experiments with the next version of the LH*$_{RS}$ prototype are in [ML02]. The current version used for the experiments below builds upon that one. We present the prototype in greater detail in [LMS04], as we have shown it to the public at VLDB-04. Further details, 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 the Windows 2000 Server. Internally, each client and server processes queries and data using threads. The threads communicate through queues and other data structures and synchronize on events. We use mainly 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. The *working threads* (currently four) process simultaneously queries and data, whether received or send out.

The communication uses the standard UDP and TCP/IP protocols. Clients communicate with servers through UDP, except when the data records are larger than a datagram could be (64 KB). A listening thread timely unloads the UDP buffers to prevent loosing a datagram. There is also a flow control mechanism. The servers communicate using TCP/IP for data transmission during the bucket split or recovery, and UDP for other purposes. 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 a larger number of incoming requests more effectively. In fact, it skips the connection dialogue that previously was necessary for each request. It proved to be by far the most efficient connection mode in our experience with LH*$_{RS}$.

Many measures of the operations using the parity calculus reported below compare the use of $GF(2^8)$ and of $GF(2^{16})$. We expected the latter to be faster. Experiments mostly (but not always) confirmed our intuition and quantified it. $GF(2^{16})$ provided the most noticeable acceleration for the erasure correction. We could also confirm and measure the benefit of using our logarithmic matrix **Q**, derived from our newest version of a parity matrix **P**, with a first column and a first row of ones. We then measured the speed of the operations involving the parity updates, namely the inserts, file creation with

splits, updates, as well as bucket and record recovery. The study varied the availability level from 0 to 3. We left the study of delete, merge and scan operations for the future. As we already said, the first two operations are of lesser practical interest, whereas the latter is independent of the parity calculus unless it triggers a record recovery. We also measured the speed of key searches as basic reference. We averaged each measure over several independent experiments.

Practical considerations lead to simplified implementation of some operations. Also, the experiments modified our own ideas on the best design of some operations. We discuss theses issues in the respective section.

The test-bed 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. We used the latter as a client. Others served as data and parity servers. Sometimes, we also used additional client machines (733 MHz, P3). Our network was a 1 Gbps Ethernet.

5.2 Parity Generation

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 a 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 investigated the influence of the column and row of ones in the parity matrix **P**, or equivalently, of a column and row of zeroes in **Q**. This means that updates to the first data bucket only involve XORing, but no Galois field multiplications. For $GF(2^8)$, the processing time shrank 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. Again, $GF(2^{16})$ was always faster, but by only about 15 %.

As expected thus, the **Q** with first column and first row of zeroes yields the fastest encoding. We therefore used only this choice for our experiments below. We attribute the always higher savings for $GF(2^8)$ above to the higher efficacy of XORing byte sized symbols.

5.3 Key Search

The key search times in the normal mode serve of the referential to the access performance of the prototype, since they do not involve the parity calculus. We measured the time to perform random *individual* (synchronous) and *bulk* (asynchronous) successful

key searches. All measurements were done at the client. We start the timing of an individual search 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 reported is the average over a synchronous series of individual searches, i.e., one starting after the end of another. We start the clock to measure a bulk search when the client gets the first key from the application and we stop it when the application receives the last record searched. We report the average time. During the bulk search, the client launches searches asynchronously, as usual for a database query. These searches use UDP and a custom flow control method to prevent a server overload. Interestingly perhaps, most of our experiments did not even trigger this mechanism.

We measured the search time in a file of 125,000 records, distributed over four buckets and servers. A record had a 4 B key and 100 B of non-key data. The average individual search time was 0.24 ms. The bulk one was 0.06 ms, i.e. four times faster. The individual search is thus about 40 times faster than a single disk access, whereas the bulk one is about 200 times faster. The server CPU speed was the limiting factor for the former and the speed of the client for the later.

5.4 Insert

We timed series of individual and bulk inserts in the normal mode. The inserts addressed bucket 0, without triggering any bucket split. The idea was to test the most unfavorable scenario, in which all the inserts from a client end up at a single bucket. Inserts into multiple buckets with splits gave rise to different experiments, Section 5.5. We defined the individual insert time for this experiment as starting when the client receives the record from the application and ending when the client synchronously gets the acknowledgement from the data bucket. Bulk inserts used asynchronous acknowledgements for the flow control, as for the bulk key searches. The client acknowledges the insert to the application after it gets its own acknowledgement. An overflow of its queues could result otherwise.

We did not measure the degraded mode. It would amount to collect arbitrary timeouts. We only implemented 1PC, leaving the more complex 2PC protocol for the future. As we mentioned already, some, perhaps many, prospective implementations are also likely to choose 1PC only. We have nevertheless made additional measurements, forecasting the 2PC performance as well. Our experiments for the inserts measured the basic case of the data bucket sending the acknowledgement to the client immediately after the messages to its k parity buckets. Using 2PC would not change the insert time in this conditions, as long as the additional load did not saturate the server.

We timed a series of 10,000 inserts into an initially empty bucket of b = 10,000, avoiding thus the split, as wished. Again, a record consisted of a 4 B key and 100 B of non-key data. We recorded 0.29 ms for k = 0, 0.33 ms for k = 1 and 0.36 ms for k = 2. The choice of *GF* did not matter as the Δ-record is simply the inserted one. The average bulk insert time was 0.04 ms, i.e., seven to nine times faster. That time was the same as for updates, discussed in Section 5.6 below. Both bulk times were measured in the same way, and are similarly independent of k.

The figures above show that adding the first parity bucket to a 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 %. The increment is due mostly to the additional message. Its cost is about 0.03 ÷ 0.04 ms, as the bulk insert time shows as well. This let us to infer trivially the timing for larger k's.

Table 1 confirms these calculations for the updates. It also shows that the time for the message with the Δ-record followed by its acknowledgment was in those experiments about 0.05, i.e. about the double of the time above. These numbers allow us to estimate the total time of an insert processing at the data bucket and the throughput if 2PC was used. Related details are in Section 5.6. Applied here, for instance to k = 3, they show this time of about 0.36 + 4*0.03 = 0.48 ms. The rationale for the formula is the time for the last acknowledgment yet to come in from the server (others came normally earlier, in parallel to the outgoing messages from the client), followed by three commit messages sent in order by the client. Accordingly, we forecast the throughput as about 2000 inserts/s. For k = 2 the time would be smaller, about 0.42 ms, and the throughput higher, reaching about 2400 inserts/s. Etc.

All this appears a quite efficient behavior. Notice finally that the measured insert times at the client are respectively at least 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 time was bound by the maximal client speed.

5.5 File Creation

Figure 7 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 we give the numerical values 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 ms and of 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 are at least 20-30 times faster than disk accesses.

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. We see now a cost of the parity calculus 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%. The increase is caused mainly by the additional messaging at the data bucket.

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 a rate of 0.33 MB/sec for k = 0, 0.25 MB/sec for k = 1, and of 0.23 MB/sec for k = 2. These numbers allow us to predict linear creation times for other record sizes. Bulk creation times and rates should be limited by the client and yet be about ten times faster from the application's point of view. For instance, less than a minute should suffice for a 1 M record file.

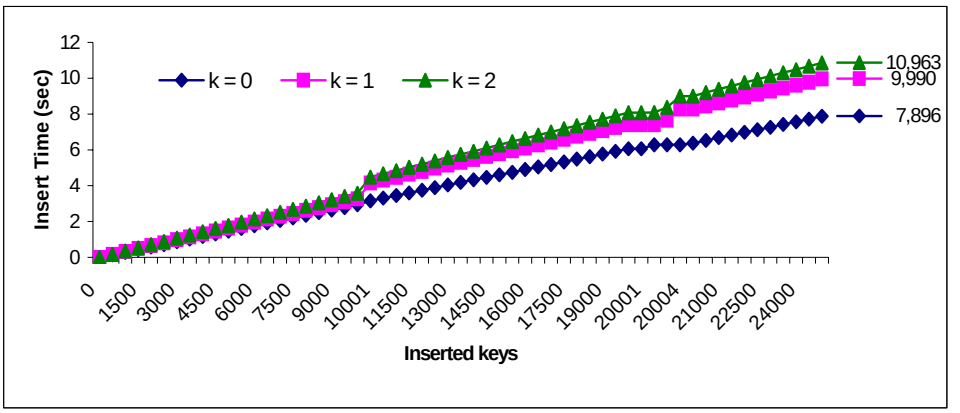

Figure 7: Creation times (seconds)

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** speeds up encoding time by almost 2 %. While the acceleration appears to be slight, we recall that it comes at no cost.

5.6 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 the caching. Table 1 shows the results for bulk and individual updates. All updates used UDP and 1PC. As before, we only measured 1PC. This time, however, the data bucket waits for all 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. In addition, if the series is longer, then more records are perhaps temporarily stored in the queue of the listening 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. Nevertheless, 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.

The other columns list the average individual update times for $k = 0...3$. The bulk update times are basically six times faster, as the comparison for $k = 0$ shows. The numbers show also that using one parity bucket doubles the update time into the data bucket alone. This result matches the intuition. However, adding more parity buckets only increases the time by 10% to 20%. Notice that adding the 3rd parity bucket adds only 2 - 7 %. All this is again nice behavior. One may further extrapolate these results to the server 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.

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

Compared also to the inserts, the bulk times do not change, as the client processes inserts and updates at the same possible speed. The individual update time takes in contrast considerably longer. The times in Table 1 for $k = 1$ already is almost 45 % longer than the time to insert. It is the measure of the additional processing of the Δ-record (XORing with the before image) followed by the UDP messaging and the waiting for the acknowledgements. The first ones come back in parallel to the outgoing messages, hence only the last one actually counts. To speed up the parallelism, the listening thread at the parity buckets acknowledges the Δ-record at the earliest moment, namely when it saves it in its queue from the communication buffer, before the actual encoding into the parity record. That is why the times to process the message from the bucket followed by its acknowledgment are only about twice of the cost of a message, i.e., about 0.05 ms on the average. This matches the results for the inserts where the measured message time was about 0.03 ms. The results show furthermore that the XORing at the client took about 0.18 ms on the average. It appear slightly slower for GF (2^{16}). This reinforces the similar findings in Section 5.2.

The individual insert time for $k = 0$ is about 15 % longer than that of an individual update. This is the price for the internal LH splits within the bucket for the inserts. Next, if the updates used 2PC, then the data bucket would need to issue additional k commit messages. An update time for, e.g., $k = 3$ and GF (2^{16}), assuming finally 0.03 ms per message would need at the data bucket about 0.59 + 0.09 = 0.68 ms. Hence the throughput would be almost 1500 updates/s. This is less than for the inserts, but the number still appears rather very attractive by comparison to the present disk files even for $k \leq 1$ only.

5.7 Bucket Recovery

The recovery manager performs this manipulation as in Section 4.1. For implementation related reasons however, 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 one, two, and three buckets, made unavailable. We neglected the synchronization phase whose influence on the response time would be minimal anyhow. The recovery manager loops conceptually over all the existing record groups, i.e., over all the parity records in the parity bucket (Section 4.1). 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 8 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 $GF(2^{16})$. Since the operation is much longer than those of individual records discussed till now, we measured not only the total time (T), but also the process time (P), and the communication time (C).

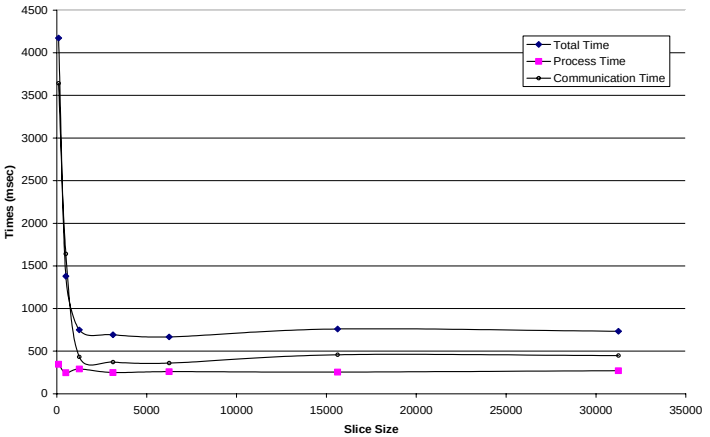

Figure 8: A single data bucket recovery time (milliseconds) as function of the slice size *s*.

We determined 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 8 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 based on the already mentioned passive TCP connections. The result means 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 show the clear superiority of our current implementation.

Table 2 completes Figure 8 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 value of $P + C$ is thread synchronization and switching time. We have measured all these times also for the other *s* values in Figure 8. For $s \geq 1250$, the differences to the times listed here were under 15 % for 1-DB (data bucket) recovery, 5 % for 2-DB recovery and 2 % for 3-DBs. The first line of the table presents the 1-DB recovery using the XOR decoding only, as in Figure 8. The second line shows 1-DB recovery using the RS decoding (with the XORing and multiplications). We used another parity bucket instead of the one with ones only, just as in [LS00]. The XOR calculus showed itself to be 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 as it contains the *C* value. This value is naturally rather stable and turns out to be relatively important with respect to *P*, despite our fast 1 Gbps

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)	15625	0,520	0,225	0,296	6250	0,552	0,240	0,312
1-DB (RS)	6250	0,932	0,630	0,297	15625	0,656	0,354	0,297
2-DBs	15625	1,464	1,156	0,302	15625	0,875	0,562	0,281
3-DBs	6250	2,094	1,711	0,374	15625	1,188	0,823	0,361

Table 2: 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 0,520 s to recover 1 DB in our experiments, and less than 1.2 s to recover 3 DBs, i.e., 9.75 MB of data in three buckets. The growth of *T* appears to be sub-linear with respect to the number of buckets recovered. This is a consequence of parallelism at the implementation level, and of the recovery of a bucket group as a whole at the conceptual level. The numbers convincingly confirm the advantage of using $GF(2^{16})$. It halves *P* of any recovery measured, with the exception of recovery using XOR only. This was the rationale for our choice of this field for the basic LH*$_{RS}$ scheme, given also that with it parity calculation is as fast as using $GF(2^8)$. Notice that *C* in Table 2 increases more moderately than *T* as the function of the number of DBs recovered.

As discussed in Section 3.4, we used the logarithms of the coefficients in $\mathbf{H}^{-1}$ to obtain the recovery times in Table 2. We also experimented using $\mathbf{H}^{-1}$ directly. The results for *P* were slower, up to 10 % for 3-DB recovery, confirming our choice.

The flat character of charts in Figure 8 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 5.89 MB/sec of data. Next, we recover two data buckets of the group at the rate of 7.43 MB/sec. Finally, we recover the group from 3-unavailability at the rate of 8.21 MB/sec. If we have thus for instance 1 GB of data per bucket, the figures imply a value for *T* of about 170 sec for 1-DB recovery, 270 sec for 2 GB recovered, and 365 sec, i.e., about 6 min per 3 GB recovered, 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.

	$GF(2^8)$			$GF(2^{16})$		
	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 3: Parity bucket recovery times (seconds) for the slice size of s = 31 250 records.

Table 3 presents the parity bucket recovery time, again for 31,250 records to recover and s = 31,250. The time T to recover bucket P0 using XOR only, analyzed in the row marked PB (XOR), is faster than for the other buckets using the RS calculus. We observe again fast performance. The XOR only recovery using 2 B symbols (i.e. from $GF(2^{16})$) is less efficient than that using 1B symbols (from $GF(2^8)$). We have a reverse picture for the other parity bucket, as the last row in Table 3 shows. The small difference in P value with respect to the one reported in Section 5.2 is due to the experimental nature of the analysis. The measurements naturally vary slightly among experiments. Similarly as for data buckets, Table 3 allows us to infer 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.

5.8 Record Recovery

Our prototype places the record recovery manager at one of the parity buckets. It acts as described in Section 4.2. Table 4 shows the average total record recovery time T we measured. The bucket size was b = 50 000. The group size was again m = 4. The times are measured at the parity bucket. They start when the bucket gets the message from the coordinator, and end with 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, we measured the average scan time of our parity bucket to locate the key c of the data record, as described in Section 4.2, to be 0.822 ms. This is the dominant part of the total time (62% and 64% respectively).

The results match our intuition and the experimental key search times. They confirm that the basic record recovery capability should prove to be often sufficient in practice. The deterioration of the search time with respect to the normal one, we recall of 0.24 ms, is nevertheless about 5.5 times. If one seeks faster record recovery, or if buckets are much larger, the additional index (c, r) at the parity bucket, mentioned in Section 4.2, should help. For $GF(2^{16})$ and our first parity bucket one may estimate the decrease to almost 1.296 – 0.822 = 0.474 ms. The ratio to the normal time becomes less than twice. Notice however the price tag for the index: more storage at the parity bucket and additional

processing of an insert and delete at the parity bucket. Knowledge of the scan time allows us further 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 at least about eight times.

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

Table 4: Record recovery times (milliseconds)

6 RELATED WORK

Traditionally, high availability was not part of a (key-based) data structure in both centralized and distributed environments. If needed, a lower storage level such as mirroring or RAID like techniques provided it. This approach simplifies the design of a data structure, but it can deteriorate access times in a distributed environment. For example, a dictionary data structure using hashing could place a data unit at some particular node. However, the underlying RAID system could move the data to a different node or even distribute it over several nodes. This lower level interference would result 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 in their internal structures, e.g., the

bucket size. In any case, the two files in LH*$_M$ are more strongly coupled than usual mirrors. LH*$_M$ can even recover some cases of double or more unavailability.

[L&al97] proposed another 1-availability SDDS called LH*$_S$. LH*$_S$ partitions a record into *n* segments, stored at *n* different sites. It adds an $(n+1)^{st}$ XOR parity segment at some other site. Compared with LH*$_M$, the parity overhead is much smaller, close to 1/*n*. 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 generalizes 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 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 record 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 in the group. 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. A deeper comparison of trade-offs between LH*$_{SA}$ and LH*$_{RS}$ remains to be undertaken.

Outside the domain of SDDSs, research has addressed high-availability needs for distributed flat files for many years. The dominant approach was replication, [H96]. The major issue was consistency of replicas, [P93]. Disk arrays in a centralized environment historically needed high availability with less storage overhead [BM93], [H&a94]. The

arrays typically have 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 to be 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*-available (*k*>1) erasure correcting codes discussed in Section 13.5 (Appendix C) came from this context.

High-availability is also a general goal for a DBMS. Nevertheless, our aspect of this concept, 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 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 to be the most efficient in the domain, [S99]. It hashes partitions of a table into fragments located each at a different node. The nodes communicate using a dedicated high-speed switch. Clustra hashing is static, hence has 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), 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 one-available and limits their 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 grid computing and very large Internet databases. Some simple techniques are already in everyday use. They are apparently replication based, but covered by corporate secrecy. 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 those based on Wi-Fi, also have 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 dispersion can also be larger by orders of magnitude, possibly running into hundreds of thousands in the near future and later reaching millions, spread worldwide. This thinking clearly shares some rationales for LH*$_{RS}$. Our scheme could thus turn out to be useful for these new applications as well.

7 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. 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 from our initial proposal, [LS00]. Changes were made to the parity calculus and to various algorithmic issues to make the file always at least $(K - 1)$ – available and make the parity calculus as fast as possible. We thus have increased the Galois Field size to $GF(2^{16})$. We have changed parity matrix **P** so it has a first column and first row of ones. We have also improved the calculus by taking advantage of logarithmic parity. We have built a prototype implementation that proves 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 showed a substantial speed-up of various operations.

At present, for the most frequent case of $k = 1$, our erasure correction uses XORing only and is thus as fast as possible to our best knowledge, under our constraints, that of storage efficiency especially. For $k > 1$, it appears more effective in practice than if we used any alternative parity code or scheme we are aware of. This is also true for our own earlier approach, as we just mentioned. The yet unique presence of the row of ones contributes to this performance. Likewise, our original use of the logarithmic matrices. 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 any application.

Finally, our prototype implementation has shown very fast access and recovery performance. Our test-bed 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 many times faster. This performance is partially due to processing data in the distributed RAM. All together, the capabilities of our scheme should attract numerous applications of hash files, for which LH*$_{RS}$ should be "plug-compatible", as we discussed in the introduction. To recall, these applications are potentially very numerous, as hash files are ubiquitous. The area includes new domains of grid computing and of P2P, and popular DBMSs. The latter still use the more limited static and 1-available replication or RAID storage for high-availability.

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 discussed variants, including those in Appendix C .

ACKNOWLEDGMENTS

Many thanks to Jim Gray and to the anonymous referees for helpful suggestions.

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*, p. 62 – 72.

[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/.

[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.

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

[BM03] Ben-Gan, I., Moreau, T. Advanced Transact-SQL For SQL Server 2000. Apress (publ.), 2003. ISBN 1-8931115-82-8.

[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).

[CERIA] CERIA Home page: http://ceria.dauphine.fr/

[CRP96] 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 2003.
[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).
[G02] Gray, J. & al.. Data Mining of SDDS SkyServer Database. Intl. Workshop on Distributed Data Structures, (WDAS 2002), Carleton Scientific.
[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.
[LMS04a] LITWIN, W., MOUSSA, R., SCHWARZ, T., LH*$_{RS}$: A Highly Available Distributed Data Storage System. Research Prototype Demonstration. VLDB Toronto 2004.
[LMS04b] LITWIN, W., MOUSSA, R., SCHWARZ, T., LH*$_{RS}$: A Highly Available Distributed Data Storage System. CERIA Technical Report, December 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, 150-159.
[MS97] MACWILLIAMS, F. J., SLOANE, N. J. A. The Theory of Error Correcting Codes. Elsevier/North Holland, Amsterdam, 1997.
[M03] MOUSSA, R. in Distributed Data and Structures 4, Carleton Scientific, (Records of WDAS 2002, Paris March 2002). 2003.
[M04] MOUSSA, R. Experimental Performance Analysis of LH*$_{RS}$. CERIA Res. Rep. [CERIA].
[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.
[R98] RAMAKRISHNAN, R. *Database Management Systems*. McGraw Hill, 1999.
[RFC793] RFC 793 - Transmission Control Protocol 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.
[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), 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)*, 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.

Received August 2004; revised January 2005; accepted May 2005.

APPENDIX A PARITY MATRICES

We present the first 32 rows by 10 columns submatrices of the generic parity matrix **P'** and of the generic logarithmic parity matrix **Q'** for $GF(2^{16})$ that 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, values that should suffice in practice. Next, we show 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. The entries of **Q'** are given in decimal as logarithms numbers between 0 and 254. The program to generate the complete matrices can 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 9 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 10 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 11: 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 12: 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	$0 — i$
N	current number of data buckets in the file	$N = 2^i + n$
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	
P	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 5
antilog (i)	antilogarithm of integer i ; $0 \leq i < 2^f - 1$	Table 5
I	identity matrix m x m	
P'	generic parity matrix	Figure 9
P	actual parity matrix	upper left m x K submatrix of **P'**
Q'	generic logarithmic parity matrix	Figure
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

ELECTRONIC APPENDIX C

References not in the main text are given at the end of the appendix. We use curly brackets {} to identify them.

8 LH* ADDRESSING ALGORITHMS

The LH*$_{RS}$ *Forwarding Algorithm* executed at bucket *a*, getting a key-based request from a client is as follows, [LNS96]. Here *c* is the key, and *j* is the level of bucket *a*. As the result the bucket determines whether it is the correct one for *c*, according to (LH), or forwards the request to bucket *a'* > *a*.

(A1) $a' := h_j(c)$;

if $a' = a$ **then** accept *c* ;

else $a'' := h_{j-1}(c)$;

if $a'' > a$ **and** $a'' < a'$ **then** $a' := a''$;

send *c* to bucket *a'* ;

The *Image Adjustment Algorithm* that LH*$_{RS}$ client executes to update its image when the IAM comes back is as follows. Here, *a* is the address of the last bucket to forward the request to the correct one, and *j* is the level of bucket *a*. These values are in IAM. Notice that they come from a different bucket than that considered in [LNS96]. The latter was the first bucket to receive the request. The change produces the image whose extent is closer to the actual one in many cases. The search for key *c* = 60 in the file at Figure 1b illustrates one such case.

(A2) if $j > i'$ **then** $i' := j - 1$, $n' := a + 1$;

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

9 GALOIS FIELD CALCULATIONS

Our *GF* has 2^f elements ; $f = 1,2\ldots$, called *symbols*. Whenever the *size* 2^f of a *GF* matters, we write the field as $GF(2^f)$. Each symbol in $GF(2^f)$ is a bit-string of length *f*. One symbol is *zero*, written as 0, and consists of *f* zero-bits. Another is the *one* symbol, written as 1, with *f*-1 bits 0 followed by bit 1. 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 $f = 16$ turned out to be even more practical and became our final choice, Section 5. 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 call this *GF 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 5 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 ⊕ 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*^ *b*. The multiplication and division are more involved. There are different methods for their calculus. We use a variant of the *log/antilog* table calculus [LS00], [MS97].

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 5: Logarithms for *GF*(256).

The calculus exploits the existence of *primitive* elements in every *GF*. If α is primitive, then any element $\xi \neq 0$ is α^i for some integer power *i*, $0 \leq i < 2^f - 1$. We call *i* the *logarithm* of ξ and write $i = \log_\alpha(\xi)$. Table 5 tabulates the non-zero $GF(2^8)$ elements

and their logarithms for $\alpha = 2$. Likewise, $\xi = \alpha^i$ is then the *antilogarithm* of i and we write ξ = 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:

$\xi \cdot \psi$ = antilog(log(ξ) + log(ψ) mod (2^f–1)),

ξ / ψ = antilog(log(ξ) – log(ψ) + 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 5 shows the logarithms for F.

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

Figure 13: Galois Field Multiplication Algorithm.

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 14 gives the pseudo-code generating our `log` and `antilog` multiplication table (Table 5). Figure 13 shows our final multiplication algorithm. We call them respectively `log` and `antilog` arrays. The following example illustrates their use.

Example 7

$45 \cdot 1 + 49 \cdot 1a + 41 \cdot 3b + 41 \cdot ff$

$= 45 + \text{antilog}(\log(49)+\log(1a)) + \text{antilog}(\log(41)+\log(3b)) + \text{antilog}(\log(41)+\log(ff))$

$= 45 + \text{antilog}(152 + 105) + \text{antilog}(191 + 120) + \text{antilog}(191+175)$

$= 45 + \text{antilog}(257) + \text{antilog}(311) + \text{antilog}(191+175)$

$= 45 + \text{antilog}(2) + \text{antilog}(56) + \text{antilog}(111)$

$= 45 + 04 + 5d + ce$

$= d2$

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 5. We add these up as integers to obtain 257. This value is not in Table 5, 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 5 it contains 51 (in hex), which is the final result.

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

Lemma for Section 3.2**.** Let *m* be the current group size and *m'* be the generic group size. Denote the generic parity matrix with **G'**. Form a matrix **G** as in Section 3.2 from the first *m* rows of **G'**, but leaving out the resulting zero columns *m*+1, ... *m'*. Then we obtain the same encoding and decoding whether we use **G** and the first *m* data buckets, or whether we use **G'** and *m* data buckets but with all but the first *m'* data buckets virtual zero buckets.

Proof: Consider that for the current group size $m < m'$. There are $m' - m$ dummy data records padding each record group to size *m'*. Let **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(**H'**) = det(**H**)·det(**I**) = det(**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.

11 ADDITIONAL LH*$_{RS}$ FILE MANIPULATIONS

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

11.1 File Creation and Removal

The client creates an LH*$_{RS}$ file *F* as an empty data bucket 0. File creation sets the parameters *m* and *K*. The latter is typically set to *K* = 1. The SDDS manager at bucket 0 becomes the coordinator for *F*. The coordinator initializes the file state to (*i* = 1, *n* = 0). The coordinator 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.

11.2 Key Search

In normal mode, the client of LH*$_{RS}$ searches for a key using the LH* key search, as presented in Section 2.1.2. The client or the forwarding server triggers the degraded mode if it encounters an unavailable bucket, called a_1. It 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. In addition, it performs record recovery.

11.3 Scan

A scan returns all records in the file that satisfy a certain query Q in their 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.

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 8

Assume that the file consists of 12 buckets 0, 1, … 11. The file state is $n = 4$ and $i = 3$. Assume also that the client 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.

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 4.1. Once they are all completed, the coordinator sends the scan to the recovered data buckets. The client waits until the scan completes in this way. Whether the termination is normal or degraded, the client updates finally its image and perhaps the location data.

Example 9

For change, we consider now a file in state $(n, i) = (0,3)$, hence with 8 buckets 0,1…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.

Assume 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.

11.4 Delete

In the normal mode, the client performs the delete of record c as for LH*. In addition, the correct bucket sends the **Δ**-record, the rank r of the deleted record, and key c to the k parity buckets. Each bucket confirms the reception and removes key c. If c is the last actual key in the list, then the parity bucket deletes the entire parity record r. Otherwise, it adjusts the B-field of the parity record to reflect that there is no more record c in the record group.

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

11.5 Merge

Deletions may decrease the number of records in a bucket under an optional threshold $b' << b$, 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.

12 PERFORMANCE ANALYSIS

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

12.1 Storage Occupancy

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

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

$\alpha_g = \alpha_d / (1 + k / m)$.

The average load factor α_f of the file depends on its state. As long as the file availability level *K'* is the intended one *K*, we have $\alpha_f = \alpha_g$, provided $N >> m$ 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 $\alpha_g (K)$. This is the highest possible load factor for the availability level *K* and α_d. We thus achieve the highest possible α_f for any technique added upon an LH* file to make it *K*-available.

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

The record group size *m* limits the record and bucket recovery times. If this time is of lesser concern than the storage occupancy, one can set *m* to a larger value, 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. However, the obvious penalty are about l times greater messaging cost for bucket recovery, since m buckets have to be read. Fortunately, this does not mean that the recovery time also increases l times, as we will see. Hence, the trade-off can be worthy in practice.

Example 10

We now illustrate the practical consequences of the above analysis. Assume $m = 8$. The parity overhead is then (only) about 12.5 % for the 1-availability of the group, 25 % for its 2-availability 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. However, unavailability of buckets 8-10 is catastrophic. Consider then rather the choice of $m = N_1 = 16$ for the file starting with $K = 2$. The storage overhead remains the same hence is α_f. Now, the file tolerates that unavailability as well, even that of up to any four buckets among 1 to 16.

Consider further the choice of $m = 256$ and of $N_1 = 8$. Then, $K' = 1$ until $N = 16$, $K' = 2$ until $N = 128$, then $K' = 3$ 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 %.

12.2 Messaging

We calculate the messaging cost of a record manipulation as the number of (logical) messages exchanged between the SDDS clients and servers, to accomplish the operation. This performance measure has the advantage of being independent of various practical factors such as network, 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 6 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.

Manipulation	Normal Mode (N)	Degraded Mode
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 6: Messaging costs of an LH*$_{RS}$ file.

To evaluate bucket recovery cost in this way, we follow the scheme in Section 4.1. A client encountering an unavailable bucket sends a message to the coordinator. The coordinator responds by scanning the bucket group, receiving acknowledgments of survivors, selecting spares, receiving acknowledgments from them, and selecting the recovery manager. This gives us a maximum of $3+2m+3l$ setup messages (if l buckets have failed). Next, the recovery manager reads m buckets filled 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.

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

The other costs 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 assume that l unavailable buckets found are each in a different group. The coefficient B_1 denotes the recovery cost of a single bucket. Several 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.

13 VARIANTS

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

13.1 Parity Messaging

Often, an update changes only a small part of the existing data record. 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 alternative design sends the Δ-record only to the first parity bucket, and without a rank. The first parity bucket assigns the rank. It is also in charge of the updates to the *k*-1 other parity buckets, if there are any, using 1PC or 2PC. The drawback of the variant 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, only necessary for updates.

13.2 Availability Tuning

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

More precisely, to decrease the availability of a group from $k > 1$ to $k–1$, it suffices to delete the 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 some of the alternative coding schemes 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 read efficiently data buckets in parallel. Their efficiency remains for further study. 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 for the new bucket. For instance, the data buckets may be made aware of the existence of this bucket before it requests the first data records. As 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.

13.3 Group Size Tuning

We recall that the group size m for LH*$_{RS}$ is a power of two. The group size tuning may double or halve m synchronously for the entire file, one or more times. The doubling merges two successive groups, which we will call *left* and *right* that become a single group of $2m$ buckets. The first left group starts with bucket 0. Typically the merged groups 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.

13.4 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 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 $j = (r + i)$ mod m. Do 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 happens, 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 $N = 1K$ it drops in fact to zero, since the last group is the right one. The file remains 2-available.

Partitioning should usually also shrink the recovery time. The recovery operation can now occur in parallel at each parity bucket. The time for decoding the data records in the *l* unavailable data buckets is then close to *l*/*m* fraction of the basic one. In our 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 $n >> 1$.

13.5 Alternative Erasure Correcting Codes

In principle, we can retain the basic LH*$_{RS}$ architecture with a different erasure correcting code. The interest in these codes stems first from the interest in higher availability RAID [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 constant interest in new erasure correcting codes. These may compare favorably with generalized Reed-Solomon codes. (In addition, decoding Reed-Solomon codes has made great strives {Su97},{GS99}, but we use them only for erasure correction for

which classic linear algebra seems best.) Nevertheless, one has to be careful to carry over the conclusions about the fitness of a code to LH*$_{RS}$. Our scheme is indeed largely different from these applications. It favors smaller group sizes (to limit communication costs during recovery), utilizes main memory (hence is sensitive to parity overhead), can recover small units (the individual record), has scalable availability, 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.

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 unavailability) proceeds at the highest speed possible. As long as k=1, any update incurs the minimal parity update cost for the same reason. In addition, for any k, updates to a group's first data bucket 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 5) 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).

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}, {Pa85}, {FHB89}. Figure 15 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 15. 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 15: 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 15 for instance, a_i, b_i, c_i with $i > 5$ at are dummy symbols. Next, in Figure 16, we have chosen $k = 2$ and $m = 3$, hence $p = 5$. We encode first four symbols from three data records a_0, a_1,..., 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 15 is the third such line.

Block array codes are linear and systematic. As for our code, we update the parity records using Δ-records. As the figure illustrates, we only use XORing. In contrast to our code 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 16 requires XORing of two symbols of any Δ-record. For instance, - the first and second symbol of the Δ-record if record $a_0,a_1,\ldots$ changes. This results in between one and two times more XORing. Decoding turns out to have about the same complexity as encoding for $k = 2$. All this should translate 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}, {BBM95}, {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 16: 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 5 show that the difference in processing using only the first 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. {Pl97} seemed to give a simpler and longer (and hence better) generator matrix for an RS code, but {PD03} retracts this statement.

REFERENCES FOR APPENDIX C

{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).*

{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.
{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.
{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.
{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.
{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.
{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.
{GS99} GURUSWAMI, V., SUDAN, M. Improved decoding of Reed-Solomon and algebraic geometry codes. IEEE Transactions on Information Theory, vol 45(6), p. 1757-1767, 1999.
{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.
{Ma00} MACKAY, D.: 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.
{Pa85} 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.
{Pl97} 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.
{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.
{Su97} SUDAN, M. Decoding of Reed Solomon codes beyond the error-correction bound. Journal of Complexity, vol. 13(1), p. 180-193, 1997.
{XB99} XU, L., BRUCK, J. X-code: MDS array codes with optimal encoding. *IEEE Trans. on Information Theory*, Vol. 45(1), Dec. 1999.