Operational concepts and methods for using RAIT in high availability tape archives

Harry Hulen
Consultant

Glen Jaquette
IBM Tucson Development

Abstract **-** In this paper we make the case that it is time to think seriously about RAIT for the largest of the large digital archives. We look at operational concepts to efficiently use RAIT for writing tapes and for reading them. We offer possibly new thinking regarding how some problems unique to tape may be helped by exploiting some less-familiar aspects of redundant arrays. Finally, we look at an operational concept to use RAIT to detect and correct hidden bit errors that are not made evident by hardware return codes or loss of access to hardware.

Definitions

RAID, like many acronyms, has become better recognized than the words it stands for. Originally meaning Redundant Array of Inexpensive Disks, this remarkable technology has been a mainstay of storage for 23 years, longer than the lifetime of today's youngest storage professionals [1]. For most of those two-plus decades, many of our colleagues have thought about, and have from time to time applied, the same concepts to tape. The term *RAIT* introduces *T* for Tape in place of the *D* for Disk, and the result, and what it should mean from a system point of view, seems intuitively recognizable. Yet, there are ramifications of using RAIT which are not obvious at first, which this paper works at illuminating.

Like RAID, RAIT is a way to aggregate physical storage volumes to create large virtual volumes, while eliminating single points of failure and in some cases multiple points of failure in the underlying physical volumes. RAIT also increases data transfer rates via striping as the data path is spread over more channels.

Here we will use the term RAIT to mean a redundant array of tapes with both striping and parity, which would be analogous to RAID-3, -4, -5, or -6. In this paper we will focus on what we will call RAIT-5, which has striping and one parity, and RAIT-6, which has striping and two parities. Where RAIT-1 might seem to apply, we use the term *mirroring*.

Throughout this paper we use the convention that the number of data tapes is designated by *d*, and the number of parity tapes is *p*. Thus the number of tapes in a redundant array of tapes is *d*+*p*.

Figure 1 shows a RAIT-6 scheme with *d*=4 and *p*=2. We must consider that the RAIT scheme, like many RAID schemes, may rotate parity fields among the tapes as each stripe is written. As an example of parity rotation, consider the state shown in Figure 1 (where the parity records are on tapes *t*5 and *t*6) as the starting point. The next stripe might have the parities on tapes *t*6 and *t*1, then tapes *t*1 and *t*2, and so on. Various rotation schemes could be used, or none at all. If none, then all *p*1 parities would be on one tape and all *p*2 parities on another. One justification for rotation is that if the data is compressible and the parities are not, then the parities are of a different length than the data, resulting in uneven tape usage if rotation was not used. Here we simply acknowledge that rotation of parities could exist and define that the notation *d* for data and *p* for parity refer to *equivalent* tapes, such that on any given stripe *d* tapes are written with data and *p* tapes are written with parity.

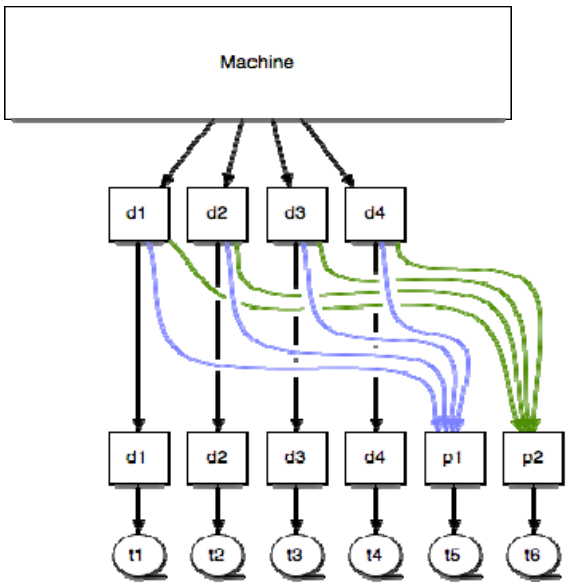

Figure 1. A (4+2) RAIT-6 scheme.

To simplify the description, we will only discuss the case where all tapes in a redundant array or in a mirrored pair are of the same generation and nominal capacity (e.g. all Linear Tape Open (LTO) fifth generation (hereafter LTO-5) tape cartridges with a nominal uncompressed capacity of 1.5 TB). It is possible to do RAIT writing across tapes of unequal capacity, but it would force more complexity and would undoubtedly be more problematic. Also some middleware such as HPSS mirrors files, not tapes, so to be precise the tapes are not mirrored. None of this affects the points made in this paper, and there is no good reason to take on this needless complexity here.

From RAID lore we borrow the terms *stripe* and *strip*. A *stripe* is a single set of records written across the tapes in the redundant array, while a *strip* is a longitudinal sequence of records written to one tape. We now use these terms to acknowledge two strategies for writing files to a redundant array of tapes:

1. *Stripe-based*. One file at a time is written across all d tapes. On retrieval, all $d+p$ tapes must be mounted if there is parity rotation. This is the method used by NCSA and the HPSS Collaboration for Blue Waters [9]. If parities are not rotated, only d tapes must be mounted unless there is an error.

2. *Strip-based*. One file is written linearly on each strip (or tape), so that multiple files are written concurrently each to a different tape. This still allows, but does not require, parity to be rotated. For example, each time a data file being written to a given tape ends, an opportunity presents itself to next write (one of the) parity streams to that tape and to write the next data file to the tape that parity had previously been written to. Because a file is written to a single tape, it can typically be retrieved by only mounting that single tape – and this can significantly improve performance in some read environments as will become apparent further down in this paper. Note that while typically a single file can be read, by mounting only the one tape it was written to, if a permanent read error is encountered when trying to read that tape, then all $d+p$ tapes which were written at once as a set must be mounted to allow recalculation of the unreadable data from the other data and parity tapes in that RAIT set.

In the remainder of this paper we are assuming stripe-based RAIT. There are very few implementations of RAIT, and the recent ones familiar to the authors are stripe-based [8,9]. The apparent reasons are that there is more precedent from RAID for a stripe-based approach, and the stripe-based approach is thought to be simpler to implement and hence more feasible. That said, future RAIT embodiments should seriously consider the read side advantages that strip-based RAIT writing offers.

The economic case for RAIT

The most obvious benefit of RAIT is economic: RAIT-5 and RAIT-6 with striping can provide greater availability of data with substantially fewer tapes than is achieved with simple mirroring. To demonstrate the saving in tape of RAIT over mirroring, let the number of equivalent data tapes in the array be noted by d and the number of equivalent parity tapes by p. For mirroring, the number of tapes would be $2d$. It follows that the number of tapes in a redundant RAIT array that would be required to store the amount of data held on a single tape would be $(d+p)/d$. For example, consider a redundant array of five tapes where d=4 and p=1. The data that a single non-RAIT cartridge would hold would require 1.25 RAIT cartridges. Simple mirroring would require two cartridges. Thus with 1.25/2 = 62.5% of the number of tapes required for simple mirroring, the redundant array achieves the same availability against a single tape failure.

With a RAIT-6 group having d=4 and p=2, the equivalent number of tapes in the redundant array required to hold the data of a single tape would be (4+2)/4 = 1.5 tapes. With dual parities, the array would have availability equivalent to three single tapes in the sense that any two tapes in the array could fail without losing data. Thus RAIT in this case would survive two failures with 1.5/3.0 = 50% of the number of tapes required by double mirroring (i.e. creating a total of 3 identical copies).

How disks and tapes are fundamentally different

Tape is notably and obviously different from disk in that tape is streaming media contained in passive removable cartridges typically maintained in and accessed via robotic tape libraries. A tape is not divided into fixed length physical sectors like disk. It is written sequentially from beginning to end within a logical partition – with that logical partition often consuming all, or at least the vast majority of, the storage capacity of the tape cartridge.

Once a given block of data is written to tape as part of a sequential series of blocks, that data cannot be updated in place without physically overwriting other data. Because of this, an erase or overwrite of data would logically invalidate all data that had earlier been written from that point to the end of the tape, and therefore overwriting is not done. Instead, when an application deletes a tape record, the record is logically invalidated, in that it is no longer pointed to, but the data on tape is not physically erased. The invalidated data is left on the tape, taking up space. The new record is written at the end of that, or possibly on another, tape. The invalidated records are left in place, and if many records become invalidated, the tape becomes sparse and inefficient. A sparse tape may then become a candidate for repacking, a process by which only the sparse valid data on a tape is read, it is written repacked to a new target tape. Once the data has been moved, the source cartridge which originally held that data can be freed and written again from the beginning with new data. Disks on the other hand have fixed-length sectors which can be rewritten or reused after the data previously stored there is no longer needed.

Both disk and tape drives can accommodate ingesting a stream of very large files efficiently and at nearly the same rates in bytes per second. Disks are also reasonably efficient for pseudo-random reading and writing of small files, with access times for high RPM Enterprise class drives on the order of five milliseconds or less – an access time that may be small as compared to data transfer time for files of any significant size. Thus, disk drives can support a high rate of

random input-output operations per second (IOPS). Tape drives, on the other hand, are very inefficient when small sets of data are read in random order, by which we mean reads are not read sequentially and are processed individually which does not allow for read optimization. Random, non-sequential reads, even if from the same cartridge, cause searches and changes in direction of tape movement. As a consequence, tape will typically give a very low IOPS in response to a random read workload, as compared to disk. Therefore, tapes are most efficient when they can be written or read sequentially and in that case can typically provide high-rate data transfers. In the case of a single IBM® LTO-5 tape drive, the native data rate (i.e. the data rate without data compression) is 140 MB/s, and if the data is 2:1 compressible the drive can transfer it at 280 MB/s.

Calculations of efficiency in reading and writing single tapes and tape arrays

In this section we use data in Table 1 to calculate efficiencies in writing and reading single tapes and redundant arrays of tapes. Much of this is summarized in Table 1, which shows data collected from IBM and Oracle web pages [3, 5] used in this paper.

Efficiency in writing tapes

In order to stream data to a tape drive at the full theoretical data rate, the goal would be to start writing data to a tape cartridge and never stop until the cartridge is full, then proceed to writing the next tape cartridge. As simple as this appears, there are challenges that impede this objective. The main one is uncertainty. Software is not always aware how quickly the next write will occur and some applications are written around the concept that they need to know to know with certainty what data has been committed to tape, as opposed to remaining exposed in the volatile DRAM buffer of a tape drive. Contributing to the uncertainty, the application may unexpectedly terminate, another user may preempt use of the tape drive, or the entire system may go down. Filling a 1.5 TB native capacity LTO-5 tape at 140 MB/s would take at least three hours, so there is a significant loss if a partially written tape is suddenly put into an indeterminant state. For this reason, some applications and middleware force a synchronization of the tape based on events such as writing a certain group of files. When a synchronization is requested, the tape drive responds by first writing the data to tape and then sending a Command Complete indicating that all data sent to the drive is now safely on tape. With most tape drives, synchronization makes it necessary to either physically stop the movement of the tape media, or to keep the tape moving and therefore create long gaps at the synchronization points. Stopping and restarting tape movement involves a cycle called *backhitch* in which the media must stop, back up to recover lost space, stop again, and then accelerate to reach the steady-state media speed required to write the next record. A backhitch can take several seconds in a modern tape drive such as LTO, and so frequent backhitches can dramatically slow tape performance. Some tape drives such as the IBM TS1130 have implemented virtual backhitch technology that allows synchronizations to be performed without having to stop tape at each synchronization. This considerably improves performance in an environment where synchronizations are frequently performed, though the tape drive has to intermittently go back and clean up what was written to recover the capacity which would be lost because the tape was not stopped [7]. It should be clear that synchronization is a challenge to streaming tape performance and should be used sparingly, even in tape drives with advanced synchronization management.

In systems where tape data is written as files to a POSIX interface, the software that chooses to synchronize each time a file is written will suffer a large performance penalty. This simple type of tape file system essentially imposes an obligation on the part of the user to write large files so that fewer synchronizations will occur. The large files may be tar files that aggregate multiple smaller files, or they may be large objects that are written as files, where an object follows a user-developed or industry-standard template. We will say more about objects in the next section on efficiency in reading tapes. At the other extreme, backup software such as IBM Tivoli® Storage Manager (TSM), or a file system such as IBM Linear Tape File System™ (LTFS) software may copy large amounts of data from memory or disk to tape at streaming speeds until end of media is reached, in which case no synchronization at all may be required until after the last record written to a tape cartridge. In between these extremes, hierarchical disk and tape systems such as HPSS and SAM-FS address the problem by automatically collecting files (including user-prepared aggregates and objects) into very large aggregates transparently to the application.

The authors suggest a best practice of keeping the number of synchronizations in writing a tape cartridge to less than a hundred. For LTO-5, 100 synchronizations would result in one synchronization occurring approximately every 1.8 minutes if they are evenly spaced. The amount of data written between each successive pair of the 100 synchronizations would average 1% of the tape length, which would be 15 GB uncompressed or 30 GB with two-to-one compression. As was stated earlier, it takes about three hours to write an entire LTO-5 tape without stopping for synchronizations, so the amount of time for 100 synchronizations would increase this by 300 seconds or five minutes, a rather insignificant increase. These calculations are linear, so that if the 100 synchronizations were increased to 1000, then the average uncompressed data size between synchronizations would be 1.5GB, and the time due to 1000 synchronizations would be about 3000 seconds or 50 minutes. The performance impact of more than 1000 synchronizations would likely be considered intolerable in a high capacity, high data rate archive environment.

Table 1. Data used in efficiency calculations

Tape hardware parameters based on an LTO-5 drive and TS3500 library

Load-to-ready time in seconds	load	12
Average block locate time from load point in seconds	locate	52
Average time to rewind to load point, seconds	rewind	48
Tape drive unload time, seconds	unload	17
Avg. library time to mount and unmount a tape cartridge	libt	12
Sustained native data rate of tape drive in MB/sec	sndr	140

Formulas used in the calculations below

ohead =	load+locate+rewind+unload+libt
dtt =	fs/sndr/de
tts =	ohead+dtt
drives =	de+pe
ttsall =	tts*drives
edr =	fs/ttsall
eff =	edr/sndr

Redundant array parameters and calculations of tape seconds, effective rate, and efficiency

Parameter description	Parameter name	Single tape	(4+1) RAIT-5	(8+2) RAIT-6	Single tape	(4+1) RAIT-5	(8+2) RAIT-6
Average file size in bytes in MB	fs	160	160	160	16000	16000	16000
Number of data elements in redundant array	de	1	4	8	1	4	8
Number of parity elements in redundant array	pe	0	1	2	0	1	2
Drive and library overhead for an average pseudo-random read	ohead	150.00	150.00	150.00	150.00	150.00	150.00
Data transfer time for one drive in MB/sec	dtt	1.14	0.29	0.14	114.29	28.57	14.29
Total tape-seconds per drive for an average pseudo-random read	tts	151.14	150.29	150.14	264.29	178.57	164.29
Total number of drives in array	drives	1	5	10	1	5	10
Total tape-seconds for all tape drives reading one file	ttsall	151.14	751.43	1501.43	264.29	892.86	1642.86
Effective MB/sec data rate per tape drive	edr	1.06	0.21	0.11	60.54	17.92	9.74
Tape efficiency as % of tape streaming native data rate	eff	0.76%	0.15%	0.08%	43.24%	12.80%	6.96%

For applications or middleware which do use synchronizations, when data is written to a RAIT set even larger objects or aggregations must be used. Assuming available stripe-based RAIT (as defined in the introduction), then to write efficiently the data units should be approximately *d* times as large as they would be when writing to a single tape, where *d* is the number of equivalent data tapes in the redundant array. Otherwise, the tapes in the redundant array will be synchronized and will experience backhitch *d* times more often than would be the case for a single tape, and this would slow the writing process significantly. To achieve efficiency in writing, a RAIT-capable storage system may create aggregates of many thousands, even tens of thousands, of user files for each write to a RAIT array.

Efficiency in reading tapes

Here we make the case that while RAIT can be very efficient for writing large quantities of data to tape, it is less efficient in reading data, particularly when reading individual files in a random order. By random order, we mean an order that does not take advantage of locality of files on cartridges and therefore may require a tape mount for almost every file read. This may also be called *pseudo-random* order because the underlying cause of the order may be deterministic, but the effect is as though it were random.

Reading a single file from an unmounted tape is a slow process because the tape must be mounted, loaded, perform a Seek and then Read, rewound, unloaded, and dismounted. The sequence of operations that occur before and after the actual data transfer takes about two or three minutes (an average time of about 150 seconds for an LTO-5 tape and a large LTO tape automation). For small files, the approximately two and a half minutes of overhead are considerably longer than the actual data transfer time, which may range from less than a second to a few seconds, depending on file size. The average file size across about 30 HPSS sites that report site information is about 160 MB [2]. This represents an average across over 200 petabytes of data. Here we use this figure as a representative average file size, and we leave it to the reader to test other sizes with the formulas provided in Table 1.

The native uncompressed data transfer rate of an LTO-5 tape drive is 140 MB/s, so once accessed the data transfer time of a LTO-5 drive is only 1.14 seconds per average-sized HPSS file, less if there is compression. The assumption of no compression is standard for scientific and imaging data, because the compressibility of such data is typically insignificant. It is also assumed that the server doing the reading has sufficient capacity to do a streamed read. For our notional average file size of 160 MB, the productive data rate reading a pseudo-randomly-selected file is about 151 seconds, or about 2.5 minutes of tape drive time to support an actual transfer of data of 1.14 seconds. We will call this 2.5 *tape-minutes* and use it as a measure of the total time a tape drive is tied up to load, seek, read, transfer data, rewind, and unload. The efficiency of the tape drive, meaning the useful benefit in this particular situation, is the actual transfer time divided by the tape-minutes of busy time, in this case 1.14/151 = 0.76%. These numbers are shown in Table 1 in the column labeled Single Tape.

For a stripe of a five-tape RAIT-5 redundant array having d=4 data records and p=1 parity record, the effective data rate drops to 0.21 MB/sec across the (d+p)=5 tapes. The total tape-seconds increase to 751, or about 12.5 tape-minutes. For a RAIT-6 configuration with d=8 and p=2, the effective data rate across the array drops still further to 0.11 MB/sec, and the total tape-seconds increases to 1501 seconds, or about 25 tape-minutes. The efficiency drops to 0.15% for a 5-tape array and to 0.08% for a 10-tape array. These numbers, which are from Table 1, are obviously too inefficient for practical use, which leads us to look at the numbers for much larger file sizes.

As a notional large file (or aggregate or object), we will consider a file of 16 GB, which is 100 times as large as the average 160 MB HPSS file and would represent an aggregate or object of 100 files of average size. Referring to Table 1, we see that for a single tape, the total tape-seconds for reading the file is 264, or about 4.4 tape-minutes. The effective MB/sec is 61, leading to an efficiency of a respectable 43%. For a five-tape RAIT-5 array, the total tape seconds is 893, or about 15 tape-minutes, and the efficiency is about 13%. For a 10-tape RAIT-6 array, the total tape-seconds are 1643, or about 27 tape-minutes and the efficiency is about 7%.

While the above discussion has compared single tape with RAIT, the single tape calculations and conclusions would also apply to mirrored tapes. Normally, only one of a pair of mirrored tapes needs to be read, whereas for a redundant array, all tapes in the array must be mounted and read for any single file. This observation further validates the observation that when compared to mirroring, RAIT-5 or RAIT-6 requires fewer tapes to write data and more tapes to read data.

Tape drive time, exclusive of the library time, can be overlapped, so that latency experienced by the user when reading files in an off-peak situation where there is no queuing is approximately the same for RAIT as for single tape. However, for the same number of tape drives and multiple non-sequential reads, the likelihood of substantial queuing increases when the assumption of RAIT is imposed, and queuing can significantly increase latency as seen by the user. Also, we can reason by example that if there are 100 tape drives available for reading files, then there could be 100 single-file reads in process at any one time. For the same 100 tape drives, if all reads were of five-tape RAIT sets, it would only be possible to read 20 files at one time, and for ten-tape redundant arrays, only 10 files could be read concurrently. The added latency due to imposing a (d+p)-wide redundant array assumption is therefore close to zero if the needed (d+p) tapes and tape drives are immediately available, corresponding to a light tape read load. However under a heavy load, the limit in our example of 10 or 20 concurrent read operations will have significant queuing compared with a non-RAIT system having 100 opportunities for concurrent file mounts. From these observations, we see that significantly more tape drives are needed to handle the total tape read workload with stripe-based RAIT sets of tapes than when writing to single tapes. It should be approximately true that a system consisting entirely of striped-based RAIT sets of tapes would require (d+p) times as many tape drives for reading files to remove file reads from a queue at the same rate as they would be without RAIT.

Strategies for improving efficiency in reading from single tapes and tape arrays

We have made the case that tape is inherently efficient to write, and it is inefficient for non-sequential random reads, which is well known by tape users. We have further shown that RAIT brings advantages of fast writes and fewer tapes when compared with simple mirroring, but is less efficient doing pseudo-random reads than single or mirrored tapes. This means that to achieve the benefits of RAIT one must plan carefully. Here we look at operational concepts for optimal use of RAIT. The first concept is obvious from the bullet item below. The rest we will explore in this section.

- Use RAIT for valuable archival data known to have a very low recall rate. (This is self-explanatory).
- Have users create large composite files which in this paper we call objects so that there will be fewer, larger reads.
- Sort files accesses by redundant arrays of tapes to reduce number of tape mounts when reading.
- Use hints and middleware familial relationships to pack related data onto fewer tapes to reduce tape mounts when reading.
- Use an operational concept we call reverse asymmetric mirroring to use a single tape a primary site and a RAIT group at a remote site.
- Combinations of these operational concepts.

Create objects at the application level.

The best way to group data for efficient reading is to organize the data at the application level with an eye toward reading efficiency. In many cases, the application is aware at write time of affiliations that are difficult to transmit to the middleware simply as hints. The use of objects, mentioned in the previous section on write optimization, can be even more useful at read time. An object, as we use the term here, consists of multiple smaller entities and often with metadata about the object in XML or similar human-readable and machine-readable format. Objects can be constructed such that there is hope that several of the data entities contained therein would be used together, so fewer tape mounts and fewer searches would be needed. An aggregate created by middleware would be less likely to have multiple files that would be retrieved together than an aggregate carefully constructed by an application. The Consultative Committee for Space Data Systems gives this example of an object, which in their terminology is an Archive Information Unit (AIU) [4]:

> An example of an AIU would be a table of numbers representing temperatures in a certain region with all the associated documentation describing how and where the temperatures were measured, what instruments were used to make the measurements, who made the measurements, why they were made, what processing has been performed on the measurements and who has had custody of these measurements since they were first created, how the measurements relate to other information, how the measurements can be uniquely referenced by others, etc.

Sort reads by RAIT group

While letting the middleware collect files into aggregates makes writes more efficient, it does little to optimize for pseudo-random reads. The most obvious way to increase the efficiency reading tapes is to reduce the number of mounts and seeks, and this requires organizing data when it is written to minimize mounts and seeks when the data is read. Reads are most efficient if data that is likely to be retrieved together is located on the same tape, and even better, if it is physically close together on that tape.

One way to impose order on a set of read requests to achieve better throughput when reading them is to accumulate and sort read requests. For example, if an application needs to read 1000 files which were written to 100 RAIT sets of $d+p$ tapes where $d=4$ *and* $p=2$, then if those requests are issued individually in unsorted order this may result in nearly 6000 tape mounts and 6000 tape demounts. But if those requests are sorted so that all files that are on a RAIT set of tapes are retrieved when that RAIT set is mounted, then each RAIT set will have to be mounted at most one time, for a total of 600 tape mounts and 600 tape demounts, which can improve overall system performance by a factor of ten for smaller files where the read transfer time is insignificant compared to the tape mount and demount time. Similarly, when a given RAIT set is mounted, the files to be read from that tape should not be read in just any order, as that can result in seek times between files on the same tape, if read in pseudo-random order, can be 30 seconds or more.

Modern enterprise tape formats including LTO-5 are serpentine formats, meaning the tape travels from beginning to end on one set of tracks, then wraps back to the beginning of the tape on a second set of tracks, then back to the end on a third set of track, and so on. LTO-5 has 80 such tracks in its serpentine. When seeking for a file on an LTO-5 tape, the drive immediately goes to the right track, and then traverses forward or backward to reach the desired file. To truly optimize the order of accessing a group of files known to be on a tape, it is necessary to know more than the order in which the files were written to tape; it is necessary to know the track each is own and the location within the track. Nevertheless, a rule of thumb from the Tucson lab is that gains can be achieved for large numbers of files by simply ordering the reads in sequential block order. This is especially true if the number of files to be read is significantly more than the number of wraps (sets of tracks written at once in a given direction for the length of tape, in the case of linear tape technology) in the tape format's serpentine..

File families

Another way to attempt to group data of similar types together is to allow the application to provide hints when each file is written, where the hints may be a project number or other file family affiliation. Files belonging to a family are then written to media identified with that family. However, such hints are a rather coarse criteria for grouping related files. If a familial group is large enough to span several tapes or several redundant arrays of tapes, then the desired reduction in number of tapes mounted for reads may be defeated. Also, if there are a great many familial groups, there may be impacts when writing to tape caused by the dismounting of a tape or redundant array of tapes associated with a familial group to allow mounting tapes for another familial group.

The main conclusion from the discussion above is that random reading of average-sized files is inefficient with conventional single or mirrored tape and is even more inefficient with RAIT strategies where files are spread across all tapes in the array. For RAIT, just as for conventional single tape and mirrored tape solutions, a good practice is to try to organize data when it is stored so that when it is read the system can perform fewer, larger reads and to keep data that is most likely to be read cached on disk.

Remote asymmetric mirroring:

Any archivist concerned about long term preservation would want to consider RAIT, most likely RAIT-6. However, the archivist would also be concerned about a catastrophic failure of an entire site. Such an archive should have data stored in at least two sites that are geographically separate. This implies mirroring data across two or more sites, which in turn raises the question of whether to use RAIT, mirroring, or both. In a hypothetical situation where cost is not a concern, one could have RAIT with dual parities at each site. Here we propose the notion of asymmetric mirroring, where one site uses RAIT and the other(s) use single tape. This concept of *asymmetric mirroring* is shown in Figure 2. If one site is in a particularly safe, remote location, it would seem wise to let that site be the RAIT site and the others, particularly the primary site, use single tape. We call the strategy *remote asymmetric mirroring*. This strategy has two important benefits for an archive that has significant read activity.

The first benefit is that the site that is most protected by geography is the site that enables read verification and forensic reconstruction of hidden errors. It is unlikely that this most protected site would be the primary site. In some actual archives, the remote site is literally a cave hewn inside a mountain. This protected remote site would be the source for a future migration of the archive to new storage technology. The protected remote site would not be the site that experiences most of the reads; that would be the primary site. Therefore the most secure site experiences less wear and tear. Instead of a lot of user reads, it would be focused on reads that were primarily administrative, to test the state of the archive.

The second benefit is that most of the productive user reads would be to the primary site, which is single tape and not RAIT. Those users with a mission to make use of the archive here and now would be reading data from a site where the data is stored on single tape. Only one tape cartridge needs to be mounted to read one file, or sometimes two if a file spans cartridges. Therefore, the remote asymmetric mirroring strategy overcomes the performance limitations of reading RAIT by avoiding reading from it for all purposes except the purpose of maintaining the long-term integrity of the archive.

RAIT concepts and facilities for data availability

Rebuilding a redundant array

In the case of RAID with parity such as RAID-3 or -5, the loss of one disk puts the entire redundant array into a degraded mode where there is no redundancy and therefore no protection against another failure. Once a new disk is made available, which is nearly instantly in the case of arrays which have a spare awaiting assignment, the storage system commences to rebuild the redundant array. To do this, the entire remaining array of disks must be read; the lost data computed using the other data and parity records, and the replacement disk re-written. While the re-build is in progress, the storage system provides service at reduced data rates. With today's terabyte-class disks, the rebuild process can take long enough to put the now-unprotected data at risk of another disk failure. If a second disk fails during the rebuild, then the data not yet rebuilt will be lost. This vulnerability is addressed by what has become known as RAID-6, which adds a second parity record to each stripe. With two parity records, the

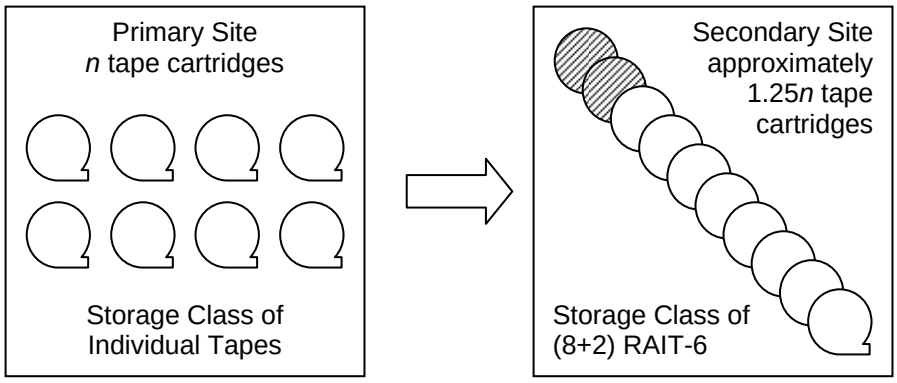

Figure 2. Asymmetric Mirroring

loss of two disks can be tolerated.

For RAIT, the failure modes are different and so is the recovery process. Tape being removable media, a failure could be a tape cartridge failure or a tape drive failure. If it is the cartridge, it could be only a stripe, or a few stripes, that are bad or it could be the whole cartridge. If the failure affects only a single file on a single tape cartridge, other files on the RAIT set may be perfectly readable and not need to be copied. The failed file would likely be copied to another RAIT set of tapes, while being reconstituted from the remaining data and parity elements. If the cartridge cannot be read at all, it would be tried on a different tape drive. If it works on the new drive, then the previous drive would be taken out of service. If the cartridge has clearly failed and not the drive, then the entire redundant array would have to be rebuilt, probably requiring it to be taken out of service until rebuilt. The rebuild would normally be a repacking process, where files would be copied to a new set of tapes. In our example of RAIT-6 with 10 tapes including the equivalent of eight tapes of data and two of parity, 20 tape drives are required to do direct tape-to-tape copying during reconstruction (10 reading the source tapes, and 10 writing the target tapes).

Real-time read verification:

Important files are usually protected with a checksum. A checksum is a fixed-size data record attached to or associated with a block of data for the purpose of detecting errors that may have been introduced during transmission or storage. Such checksums are sometimes called hash checksums, or just hashes. A checksum may be as simple as a longitudinal parity check that breaks the file into fixed-size words and XORs them together, or as strong as a cryptographic hash function such as one of the Secure Hash Algorithms of the National Institute of Standards and Technology such as SHA-256. A simple checksum has the ability to detect accidental errors with a high but not absolute degree of likelihood, whereas a strong cryptographic hash can provide near-absolute detection of any change. If the hash is stored separately (and so cannot be changed as well), the hash can protect against even clever malicious change of a file.

Reed-Solomon and other erasure codes used as RAIT-5 parity codes can be used to augment file validation that is normally done via use of checksums. If an uncorrectable read error is reported to the RAIT middleware, it would be corrected before file was even constructed and the checksum checked. RAIT therefore increases the probability of having a good copy of a file to validate with the checksum and hence a good outcome. However another level of RAIT-5 service can be provided in the form of a “read verification” that would conceptually re-compute the parity for each stripe from the data and compare it with the parity field that was read from the stripe. If equal, this would be substantial evidence that the data fields are exactly as they were when the file was written to tape. If unequal, it would mean that there is an error in the stripe and therefore some data is corrupt. It would be very likely that the file-level checksum would have detected this as well, but the RAIT verification, if performed, would discover the error first and potentially correct it. And detection at the RAIT level could be used on files that do not a checksum. In this way, the read verification utility and the checksum could discover otherwise hidden errors, called *unpointed* errors, a very rare but significant discovery. However, errors discovered by RAIT-5 read verification or by checksum confirmation would not as a general rule be correctable. It would be necessary to find another copy of the file, perhaps at a backup location. This leads us to a reason for RAIT-6.

As stated in the previous section, RAIT-6 with its two parities enables the detection and correction of a single hidden, or unpointed, error. Conceptually, an essential piece of information for efficient RAIT correction is to know which block of data is in error. Normally RAID or RAIT reconstruction would occur after another mechanism, typically at the drive level, had discovered a lost or damaged block of data within the stripe, pointing to a specific data or parity strip. Such pointed errors are treated as erasures, meaning the data known to be bad is essentially disregarded as if it were erased. Reed-Solomon codes allows correction of as many erasures as there are parity fields. RAIT-5 has one parity and so allows correction of one erasures, and RAIT-6 has two parities and so allows correction of two erasures. Thus RAIT-6 can either be used to correct two errors or detect and correct one unpointed error.

Data about undetected bit errors on tape is difficult to collect because they are exceedingly rare, and most that any that did not occur in a test environment specifically looking for this would (as the label implies) likely never be detected. Estimates of the probability of undetected tape errors are seemingly as rare as the errors. One published estimate from an Oracle specification sheet on the StorageTek automations using LTO drives [5]. This source reports a probability of one undetected error in 10^27 bits, which is about 10^26 bytes. A petabyte is 10^15 bytes, so that an undetected tape error would occur about once every 10^11 petabytes. For a very large archive ingesting 10 petabytes of data every year, the published bit error rate would lead to an expectation of one undetected tape drive read error every 10 billion years!

The 10^27 bit error rate is a calculated estimate. There is no possibility of collecting statistically meaningful data on that time scale. Calculated undetected bit error rates normally would not take into account transmission errors to and from the tape drive. They would not account for unpredictable behavior of a failing device, undiscovered firmware or software bugs, failure of people to follow procedures, and just plain bad luck. A serious archivist may therefore want to verify all data as it is read, either by both testing against a checksum and by using RAIT to correct any errors that are correctable and to detect errors which are not.

Off-line reconstruction:

As stated above, using a Reed-Solomon code two RAIT parities can be used to correct an undetected (unpointed) error. If the redundant array of tapes has two parities, it is possible to reconstruct the original file in all cases with only one error, even if it is unpointed – though the correction should then be verified by the file level checksum or hash. While it is beyond the scope of this paper to explain in detail, it is sufficient for our purposes to say that calculations are made across the data and parity bytes to determine if all the parities still check or whether they do not check and instead indicate an error. In the latter case calculations are made to determine where the errors are and the values of those errors, which allows the errors to be corrected.

RAIT-6's two parities also allow correction of up to two pointed errors (in the case of RAIT a "pointed" error would typically be a Read Permanent error from a tape drive when trying to read one of the tapes in a RAIT set). It is of course possible that there are more than two errors or that the error is so extensive that it exceeds the ability of the two parity error correction codes to correct the damage. At that point one would clearly have to look for a copy at another geographic location, as in the above discussion of mirroring strategies.

Note that correcting a single unpointed error with two parities would be rare, so while this can potentially be done in real time, on-the-fly as needed, this is not necessarily required -- the process could also potentially be done offline.

Longer time between tape migrations:

Tape cartridges are variously claimed to have a life as long as 30 years, given certain conditions of temperature, humidity, and freedom from contaminates, all reasonable restrictions [6]. Nevertheless, many serious archivists have a replacement strategy of around five years. The concern appears to be more that the tape drives will become obsolete and unavailable than that the media will deteriorate. However, there is also a motivation to pack data into more dense cartridges as they are offered so as to save tape library slots. Lawrence Livermore National Labs, for example, has reduced their total number of tape cartridges while increasing the amount of data stored by migration to higher density cartridges.

Cartridge degradation, however long that takes, is more critical than tape drive obsolescence. A tape drive can be replaced, even if it is with used or rebuilt equipment, whereas the valuable data on a tape cartridge cannot be replaced unless it can be rebuilt (e.g. via RAIT). Typically, an archive will test cartridges by reading a representative sample of cartridges on a schedule so as to detect any statistically significant degradation.

Here we propose that with RAIT-6, degradation of an individual cartridge is much less of a problem. This is because a redundant array with two parity fields can withstand the loss or degradation of up to two tape cartridges in each redundant array, and therefore the effect of any time-related degradation is reduced. Therefore while we are not offering any specific increase in time between migrations to new media, we suggest that the best practices for time between migrations can be increased, so long as the cartridge population was being monitored to assure that systematic degradation of a batch of cartridges was not occurring.

Summary

In summary, we have made these points:

- RAIT draws on a quarter century of RAID technology, including the mathematics of parity records and their use.
- RAIT can provide higher availability with fewer tape cartridges, as compared to mirroring.
- RAIT is most efficient for sequential writing and reading to tape. Because of the sequential nature of tape, writing is always sequential. So RAIT is efficient for a write-dominated archive.
- RAIT is also efficient in environments where a lot of sequential data is read each time a RAIT set is mounted.
- Stripe based RAIT is inefficient in environments where small amounts of data are read in pseudo-random order because the tape drive time required multiplies tape's unfavorable access time by the number of tapes in a RAIT set.
- Strip based RAIT promises to be significantly more efficient in an environment where data is read in pseudo-random order because in most cases it eliminates the need to mount for than one tape to access a file. Future RAIT implementations should seriously consider this alternative.
- Reads from a RAIT array should be aggregated and sorted to access all file reads needed from a given tape set at once, and those reads should be made in preferred access order, in environments which allow it, for best performance.
- Two RAIT arrays, one at each of two sites, can have data mirrored between them for Disaster Recovery purposes, or a RAIT site and a non-RAIT site can be mirrored.
- For a long term archive, geographically separated sites with use of a unRAITed single tape copy at the primary site mirrored by a RAIT-6 storage class at a safe remote site may provide the best attributes of efficient reads which are optimized at the primary site and data protection which is optimized at the remote site.
- RAIT parities can serve a secondary purpose of detecting otherwise undetected bit errors by read verification.

- To correct otherwise hidden errors found by read verification, two parities are required, vs. one parity if one depends on all errors being pointed by other error detection mechanisms (e.g. CRC across an individual block)
- RAIT-5 and mirroring (RAIT-1) allow, in theory:
  - correction of 1 pointed error, or
  - detection of 1 unpointed error – but this has almost no value in a system with a strong checksum of the files which would perform this detection anyway.
- RAIT-6 allows, in theory:
  - correction of 2 pointed errors, or
  - both detection and correction of 1 unpointed error – and this does have real value because it would eliminate the error so that the strong file checksum would now check.
- An archivist archiving files should be using strong checksums across the files and if so, then if they are further worried about unpointed errors, then there really is no reason to use anything less than RAIT-6.
- As RAIT can tolerate substantial loss or degradation of tapes without loss of data, RAIT data stores may potentially allow a longer interval between tape technology upgrades.

References

1. David A Patterson, Garth Gibson, and Randy H Katz, *A Case for Redundant Arrays of Inexpensive Disks (RAID),* Proceedings of the 1988 ACM SIGMOD international conference on Management of data
2. Harry Hulen, *High Performance Storage System Overview*, presentation slides, may be found at http://www.hpss-collaboration.org/technology.shtml
3. , Tape Library Information Center, http://publib.boulder.ibm.com/infocenter/ts3500tl/v1r0/index.jsp?topic=/com.ibm.storage.3584.doc/ipg_3584_ircc4.html
4. Consultative Committee for Space Data Systems, *Reference Model for an Open Archival Information System (OAIS)*, January 2002, http://public.ccsds.org/publications/archive/650x0b1.pdf
5. Oracle data sheet StorageTek LTO Tape Drives, may be found at: http://www.oracle.com/us/products/servers-storage/storage/tape-storage/033631.pdf
6. Imation LTO 5 data sheet may be found at http://www.imation.com/PageFiles/2510/IMT_FACTFILE_LTO5_100323_tw_D.pdf
7. Glen A. Jaquette, Paul M. Greco, James M. Karp, *Writing Synchronized Data to Magnetic Tape*, United States Patent No. 7,218,468 B2, May 17, 2007, may be found at ttp://www.freepatentsonline.com/7218468.pdf
8. Michelle Butler, *BlueWater's Archive at NCSA with RAIT*, slides from a presentation at IEEE Massive Storage Systems and Technology conference (MSST2011) May 24, 2011, and recorded on the conference web pages.
9. James Hughes, Charles Milligan, Jacques Debiez, *High Performance RAIT*, Tenth NASA Goddard Conference on Mass Storage Systems and Technologies and Nineteenth IEEE Symposium on Mass Storage Systems, Adelphi, Maryland, USA, April 2002. Hughes, the principal author, was at that time a StorageTek Fellow.