All.Net Analyst Report and Newsletter

Welcome to our Analyst Report and Newsletter

Build Backups Better (a.k.a., Back it up there buddy)

Backups and restoration of content and related metadata has been a challenge for me for much of my career. In the digital world, I have been doing this since the late 1960s, and as a result, I have, over the decades, backed up lots and lots of times, in different media, from different systems and system types, from different companies and operations, and with different levels of rigor and automation. So in some sense, what I had was a pile of semi-possibly useful pieces of content that ultimately went out of control.

Backups and restorations are hard to do well

Ever since I started using computers, I have sought a decent backup and recovery approach that would work for me. I still have not found one that really meets my needs.

For those of you at big corporates, you probably have lots of major systems to keep track of the past and get rid of older stuff no longer needed for legal purposes (something about data retention and disposition comes in here), and for those of us in smaller enterprises or homes, we have things like the Apple Time Machine, tar files, zip files, and git repositories. But all of these have problems for my use cases.

- Time Machine throws out older content when the disk runs out of space, which it often does because it keeps many copies of each file. But it has a really pretty user interface and allows you to ‘go back in time’, which is pretty cool.
- Tape ARchive (tar) files, gzipped for storage efficiency, and ‘zip’ files are very handy for transport of file structures, and they are used for backups because they have existed for long enough to be widespread and readily usable. But when you want to access content, it takes a lot of time, and things like searching are a real pain. They are intended for clod storage, not real-time access.
- GIT and similar repository solutions are designed to manage change controlled developments, differentials between version, multiple threads and consolidation of those threads into version, reversion, and more, and are very useful. Bit they are not really good backup solutions in terms of normal uses and users because of the need to systematically check out and check in different versions, which goes against the normal flow of usage, requires intentionality by the user during the lifecycle of the process, and doe not map efficiently into a usable file system for access. It is good for lots of small text files, but really bad for databases, which by the way, everything else not customized to the database is problematic for as well.
- Problems associated with identity management are also widespread. As an example, computer systems typically store content associated with ownership, groups, and so forth identified by numerical identifiers in a fixed range. The actual accessibility of content and tracking of changes is rarely kept, and because of mechanisms like roles and rules, different people associated with different roles under different rules over time mean that trying to track who actually accessed what and when comes down to logging data kept in log files usually deleted over time.

Historical variations

It is also noteworthy that historically, mainframe database systems had different types of files for databases than for other sorts of content, like text and program files, and most databases use sequential files to track all entries and changes to the database. They take snapshots in time and then have transaction files that provide step-by-step restoration of database state, often distributed across multiple locations for reliable and consistent datasets. By breaking up these sequential files or allowing the database files to operate differently, a great deal of difficulty, space, and waste in backup and restoration processes are readily feasible.

Forensics issues

In the forensics space, we seek reconstructions to generate an equivalent to the original system and content to the level of fidelity required for the matter at hand. As such, we often care about things that operational folks don't pay much attention to. For example, in many cases, sequences of events at places over times are critical to differentiation between different scenarios of what took place offered by different sides. Most backup and restoration methodologies are primarily concerned about getting access to the content in usable form and not the reliability and authenticity of the provenance information. See Appendix D and E.

Data retention and disposition

For the purposes of data retention and disposition requirements, old content without any business value is normally deleted to save time, money, and reduce liability for long-past stuff that may or may not have happened. Ignorance is bliss..

But in other cases, disposal of older data is required, for example by regulation. A good example of this is personal data associated with personnel security checks. There is an explicit requirement for not retaining old data that was the basis for adjudication processes that are beyond some previous time frame. Similarly, in legal matters, in licensing contracts, and in other similar situations, removal of old copies are built into contract requirements. In most cases, these are ignored of course, at least in terms of backups, because of the impracticality of both backing things up and keeping track of every instance of content related to every contract in place. See Appendix C for some more details.

Search and destroy – or just search

To get a sense of how problematic this is, suppose you are in a legal action related to a contract dispute where discovery requests all information in your possession or control related to an individual. The first step will likely be a preservation order telling you to preserve all the relevant content. How exactly do you find what content may be relevant? This is largely done today by search methodologies. You might search for the name of the project, the name of a person, or whatever you have for search terms. This has to cross all formats, including encrypted, zipped, tar versions of GIT repositories and email storage mechanisms. And what happens if you should find a username once associated with a person who once worked on a supply chain related to a project in a compressed email database file from 18 years ago? How do you remove it without destroying the data integrity of the whole records management system? And that assumes you can even find it in the email the refereed to JM, the initials of the individual being discussed in a meeting of other people.

Storage

Online services are out of the question for me unless I run them locally with no external network connection, because I have confidentiality requirements that, in some cases, prohibit this. I have used things like Google drive for transmitting and sharing content in various contexts, but these are things I largely use as externally accessible copies with authoritative copies kept in my systems.

I have made the decision to use file systems as a storage database wherever feasible instead of trying to put all content in large database files. This for various reasons, but very importantly, because large and ever-growing files are a nightmare for backups. Instead of large files that have to be backed up as huge blobs, I have lots of small files that have filenames associated with their dates of inception, are not overwritten but rather augmented with subsequent version files, and as such, are readily backed up by mechanisms like rsync that only back up the changed files. Of course not everybody lives in a world where they build their own systems for these sorts of tradeoffs.

By the time of publication, I hope to be able to definitively answer how many files, directories, bytes of storage, and space was and now is taken by backups. I know that in my case, it is at least 24TBytes (so far we are at about 48 Tbytes and counting). And I don't make a lot of videos or even audio recordings or anything like that. This content comes from a wide range of sources, sometimes involves other users of the systems for my companies, and at one point or another I have written, produced, analyzed, read, or applied it all.

My previous storage and recovery approach had content mostly converted from previous versions on tapes and gathered by things like tar and rsync, ported over networks and local disks over the year, copies of many things placed in multiple instances used for things like quick bring-ups of partially duplicated systems, a line of bootable CD-ROMs, specialized systems like Responder and file-system based databases, different formatted copies of emails, you name it. The lack of uniformity leads to decisions about tradeoffs. These tradeoffs mean that at best we may track changes on a daily or hourly basis, and sometimes data losses have resulted in loss of change-related information for periods of days, weeks, months or even years. This leaves gaps in time where we cannot be definitive about what took place from readily available or obvious sources.

Restorage

The fist thing I did was buy a big disk… a 24 Tbyte disk in this case. Not all that fast or fancy or expensive, but it was a starting point. Once that disk was properly formatted for use, I began copying all of the various backups I had in smaller disks

The key here is that each of the backups were restored to a different directory structure associated with its origin. For example, one directory I am going through process with is called 2004-01-01-multi-year, and it is an extract of a TGZ backup of multiple systems in use at that time which, among other things, have multiple partially redundant file structures and content used on laptops, desktops, and servers that got glommed together in 2004. It includes files with modification dates from as early as the early 1990s, possibly also from as far back as the 1970s, with some limited content from the 1960s.

As it turns out, I cannot restore all of the content at the same time on the 24 Tbyte disk, so I incrementally restore and process portions of it. I started as far back in time as was apparent

from the backups because I figured if I never finished, at least I would have the far past taken care of. I also have the notion that somehow by doing that I will be able to differentiate by the sorting of the files as they appear on my system, but that goes into another area.

Like all backup systems, and all backups, if you never restore them, you will not know that they are operable. In my case, I now run the authoritative copy of backups in ‘spinning reserves’ live disks (or in some cases SD storage) that I could and sometimes do search fairly thoroughly, but inefficiently.

Also like all backup systems of this sort, the forensic utility of the restoration is limited. We can restore the metadata associated with files and directories to get things like the creation, access, and modification dates and times, accessibility, and the content gives away a lot about them, but at this level of backup and restoration, things like the directory structures, deleted versions between backups, the order portions of files originally laid down were laid down, content past the end of file but on the media, and so forth cannot be regained in any obvious way. All we really have is the directory and file structure, metadata generated and kept, and content.

Selection and Expansion

My desire is to keep all the relevant content along with names and time frames. But there are some selection and expansion criteria that are helpful for eliminating things I really do not want to keep redundantly.

- Backup formats and mechanisms have varied over the years. For example:
  - I used MacOS and their “Time Machine” backup solution for a time.
    - As it turns out, each instance keeps a directory structure with zero-sized files for things not backed up (because they were not changed) since the last backup. This includes directories not backed up that appear as zero-sized files.
      - In my case, I simply copied only the non-zero sized everything into the backup system.
      - I also applied selection criteria as detailed below.
  - I created tar gzip files from incremental rsync backups on a daily basis, especially for remote servers. This produces a file a day containing lots of potential content, but not back-associable to the relevant content. For example, a tar file might contain the next version of a text file, but you lose traceability of the text file changes if you only consider the tar file.
    - The solution here is to untar and unzip each of the tar files to get the underlying files and file structure information and then do the rest of the backup consolidation process.
      - In my case, I knew when these started and how they were done, so I was able to act on these files fairly systematically early in the process.
      - In other cases, additional processing will be useful after initial consolidation.
  - I created tar files to move copies of file system portions
    - This is similar to the tar gzip issue, but less systematic in nature.

- Backup content sometimes included the entire boot disk, complete with the full operating system.
  - EXCEPT for reconstruction purposes, which can be valuable at times, I really do not want every copy of every version of every operating system I have ever used retained in my backups wasting time and space.
  - Disposition processes rarely cover backups, which are often in another media, only serially accessible, and difficult to process to meet deletion or selective removal criteria.
- Databases and other large files of those types often contain lots of records, the vast majority of which are unchanged over time.
  - As a result, each backup contains a unique large file with only relatively minor differences from the previous version.
    - In my case, most of these are email-related databases, and I use processes every few years to extract the emails from the datasets as monthly (e.g., mbox) files, and from there processed into separate email files, each potentially containing MIME (attachment) content.
  - There are a number of solutions to this, including approaches that take everything apart into usable portions and allow search and access within these files.
    - In my case, I chose to merely copy what was there into the consolidated backup system. A post-consolidation approach is the next step in cleaning all of this up.
  - Checkpoints and transaction records are often used in databases, and it is a reasonable selection decision to simply backup these rather than the entire database files themselves from each instance. If you have done this in the backup process, that's great. But for those of us that have not…
    - One option is to treat the databases as sets of records and extract the records for consolidation. Of course the problem is that you can have lots and lots of records, each represented by a tiny file, and it consumes the space with relatively little ultimate utility.
  - Ultimately, this comes down to tradeoffs between space, time, and utility for different purposes. And these are business decisions.

Consolidation

Here's the thing. I want to keep all of the content, recognize and readily identify when and where it came from and its provenance along the path of its evolution and disparate uses, but I don't want to waste space or money, and I do want it efficiently searchable, accessible, and usable in near real-time. So the consolidation without loss of content or related structural and metadata becomes the issue.

The first step to consolidation from my perspective was the realization that copies of files could be readily identified and redundancy in storage eliminated. If I altered one byte on some other system, it would be different content at the level of files, but at that level, the difference was acceptable for the time being. So I chose the use of cryptographic hashes to identify unique instances of content. In particular, I liked MD5 checksums because they are fast

enough for practical use, there are enough different ones so that I was unlikely to encounter many hash hits at the level of a billion files (so far I have a count of about 300 million and many more to come I suspect).

However, it is known that for any cryptographic hash, there are not just the possibilities of multiple different files having identical hash valued, but more specifically, that by prepending a homing sequence to a file, you can intentionally produce a hash hit to an altered version. I have at least one such example in my file systems used for demonstration purposes in a digital forensics mock trial I participated in long ago, so I combine the MD5 checksum with the file length to produce a more likely to be unique hash for each file. See Appendix B for details.

As I go through the directory structures of the restored versions of backups, I do two things;

- For each file, I generate a hash which will be the new name of the file containing the content, and if there is not yet any such file with the same hash, I move the file into the Content area under a pathname o0f the form WX/YZ/HASH-SIZE, creating directories as needed along the way. WX copy the first 2 letters of the hash, YZ the next 2, and the hashname and size follow. The net effect is up to 65535 directories at 2 levels of hierarchy (256 files per directory).
    - Why is this important? For efficient file system use, as directories get too big, they become expensive to linearly search, and for an effective hash, the filenames should be spread across the directory structure relatively evenly. If we notion that up to 256 files per directory would remain efficient, that means about 16 million unique files would remain efficient for access in this structure. If you have more than that many unique files, you can of course create another level and got to 4 billion unique files with ongoing efficiency by adding another layer of directories, and so on.

  If there is already a file of that name, I remove the now duplicate content. In place of moved or duplicate content, I provide a symbolic link in the restored file structure to the hash-names content file, so that the directory structure looks the same, but I have eliminated the space taken by the duplicate file.
- The second thing I do is append a line to the Table of Contents (TOC) file used to track it all, that TOC file including all the desired metadata (which I generate using the stat command), the hashfile path, and the pathname of the file it comes from. I then (or subsequently) add a table of directories (TOD) containing the same metadata for all of the directories, which do not require added storage for this purpose.

Technically, at this point, I could remove the directory structures entirely, and I hope to do so in a next phase.

The next phase

Because we have a well-structured file containing the TOC (and TOD) and a set of files containing the content regardless of names, there are lots of things I can do more efficiently with this structure than I can by spanning directory trees. To make everything accessible, I can (and will some day) make a Filesystem in User SpacE (FUSE) filesystem for efficient use of this structure. At that point, there is no need to retain the original directory structure at all.

In essence, the TOC and TOD files can be put into a databases, or in my case, a set of hash files. The idea is to allow very efficient searches. For example, suppose I want to find every directory and file changed before a given date, after a given date, or between a date range, or even over a span of multiple time periods. By searching for (example) modification times in the TOC and TOD, I can create, in real-time, an apparent directory structure that has all of the components only for this selection, and present it to the user as if it were a full file system.

For things like content searches, instead of having multiple file instances to search or categorize, I only have one copy of each actual file content. From the hash file name where I find the item, I can readily identify every instance of the content across all time and systems, and provide it as a time sequence. By looking at filenames, content, and timestamps, I can make pretty good guesses as to the path and provenance of each instance and version, at the the level of fidelity of the backup metadata. And from an access perspective, I can get to it all quickly, efficiently, and transparently. See Appendix C for some more details on these.

With simple tools come major advantages. With simple data formats come major advantages. But when we start to use complex database setups to manage our backups, we start to use more and more time and space, which undoes the advantages of time and space reduction for managing backups sought in the first place. With the TOC, the content, and a few commands (like grep, sort, and awk) I can pretty quickly do most of these things above.

Post-consolidation

Depending on how much time and effort you are willing to put in for what purpose, post-consolidation processing can be very useful or time consuming (or both). All of the things not pre-processed, like large databases, tar, zip, and similar packed up sets of files or content, end up taking space, consuming operational effectiveness, and making it harder to use them without all of the software in place originally operable for such access. There are choices, and here are some examples:

- Do you want to be able to run SCRIBE on the SCR files remaining from the 1970s to be able to generate the same output as was generated at that time?
  - If so, you had better be able to reconstruct the operating environment in an emulator for the computer system type at the time, have all the binaries and libraries in place including SCRIBE itself, and then operate that for the purposes of generating the same result from the time frame. In addition, you will need to have an output device that emulates the mechanism of producing the output from the time.
    - These issues are commonly addressed in digital archival systems, requiring transformations of relevant records into forms suitable for juridical purposes over time. A common approach is to produce a new version in a now usable format, and then do i8t again as systems change over time, ultimately disposing of older content no longer desired through a post-archival reselection process.
    - In the case of SCRIBE files, it might be better to js8ut keep the source (which is text) and recreate the document in a new source format as/if/when there is a reason to do so.
  - Obviously, this is a case for horses for courses. Pick the desired action for the use case. And then when you get another use case… so think ahead.

- Do you have a use case for being able to run the operating system from the CD in the 1990s any more?
  - In my case, I have bootable CDs created in the 1990s, and I still use the images from those CDs in virtual machines for education and training purposes.
  - To run them, you need an emulator that is adequately backward compatible, or for forensic purposes, we go find or reconstruct from parts, a system of the relevant time frame, and run them from there. In patent cases, this involves, typically, 20-year old or older computer systems (hardware, software, peripheral devices, etc.)
  - In my case, we have not only the sources for everything, but also the automated build mechanisms used to create versions, and disk dump (DD) versions of the CDs, and TGZ files from within the CD that are expanded in real-time while operating, and all sorts of other ridiculous things required at the time, all of which reasonably has to be maintained as it was at the time in order to meet reasonable anticipated use cases.
- Do you need to get at the minutia of when who did what, and under what circumstances is this required?
  - When something happens and you are trying to figure out who did what and when, the original records are the most likely to be able to most authoritatively get to the answers. But this gets to record retention and disposition policies.
    - The history of that area indicates that you should keep all normal business records for as long as they server the reasonable purposes of the business.
      - There may also be legal and regulatory requirements, contractual obligations, intellectual property concerns, time and effort required, and so forth involved.
    - In my case, I chose to keep the filesystem metadata and content. I chose, for example, not to try to be able to regenerate the links between files within a file system (i.e., the inode information and connections) because my backups cross multiple instances across multiple file systems and computers and types, and seeking ways to assure that I could reconstruct this was too much effort and complexity as well as without a reasonably anticipated use case.

The more useful part of post-consolidation processing is to finish what you started when you decided to consolidate the backups, but only to do so when there is a good reason for it. For example, because the backup mechanism in this place is in a live filesystem (with replication for resiliency), I can access a .tgz file, untar (-z) it, and add the resulting extracted files to the filesystem as individual items. By doing this and using a naming convention for the resulting untarred data structure, I can retain the .tgz versions for historical utility and also have access to the content those .tgz files contain on an individual basis. For example, I could put the files in a directory called ‘untarred-from[MD5SUM-size]’ or some such thing so that the reference to the stored .tgz file is retained and differentiable while the individual file content from each only consumes space if it is unique in the consolidated filesystem. Thus the living unified backup also becomes a record of what was done with the content as the backup is used over time, subject to reconstruction of the event sequence from the records of updating the backups. See Appendix D for more details on this.

Limitations

There are also things that the structure identified above makes difficult. One example is content deletion. In deleting a file from a filesystem, you typically delete the listing in the directory, possibly overwrite the storage area to eliminate residual content, and release the storage area for reuse. But what do you do if the content could be associated with many different versions in a virtual file system? In essence, you need to either search for any other uses of the same content elsewhere in the virtual file system, or maintain a database linking all of the content to all of the names associated with it. Of course in backups, we generally DO NOT WANT TO DELETE THINGS.

Except of course for the data retention and disposition requirements. Fortunately, at least the approach described here allows for pretty straight forward search of content. Another advantage in the backup and restoration space is that backups should effectively be append-only, except when disposal is called for. As such the virtual file system can be read-only when mounted in normal (restoration) use, made append-only when adding content, and made read-write in a limited fashion for disposition purposes. To do this in a read/write filesystem and device, the underlying (Linux or whatever) filesystem can set the content and cross-reference files to be read-only, the filesystem itself can be mounted read-only, and the FUSE interface can identify them as read-only so that no attempted access will be tried, and even if this is bypassed, the files will not be alterable or delete-able UNLESS…

For the purposes of deletion, a deletion order can be created that takes the deletion requests from the user interface and, subject to review, the actual content and related pointers may be deleted during system maintenance periods. A nice twist on this is that, assuming content is the issue requiring deletion (as opposed to directory structure and naming information) the content files that have to be deleted (in all instances they are referenced) can simply have their content replaced by a standard message, like “CONTENT DELETED BY ORDER OF [reference on the provenance of the deletion request and governance information on who took the action] … ON [date and time] …” or some other standard approach. I chose to empty the content and add a reference file (with the DELETED extension). Of course the MD5 Checksum will be wrong, and future additions of identical content from other sources has to be considered, but then the perfect solution is not yet available or even fully understood.

And then there are things like the right to be forgotten and internal partial deletion of content. Because content is stored under the name of the MD5 checksum and size of the content, any alteration to content will wreck havoc on the processes in use, producing inconsistent MD5 and length checksums as well as potentially causing other failures in processing of content by applications (such as blockchain calculations with removed records).

In our case, for most of these instances, we can use the same process as for file deletion, using a file extension (REDACTED) and content message like “CONTENT REDACTED BY ORDER OF [reference on the provenance of the redaction request and governance information on who took the action] … ON [date and time] …” The original MD5 file can be replaced by a symbolic link to the redacted version, automatically updating all relevant pointers and, of course, if there are more redactions, the REDACTION file and link updated.

There are other ‘special cases’ that are likely to pup up over time, and similar solutions seem feasible for these cases as well. Ultimately, this is an experience you might be hearing more about in future articles.

Who shall backup which backups?

Of course when we do backups, we need to back up the backup and restoration mechanisms or we won't be able to restore using the restoration methods. And presumably we need to have multiple copies of the backups for reliability over time and space. I need a copy to put in my bank vault in case of facility destruction, and I usually have several live copies available in case one computer is down when I need the content. That means that deletion requests, updates, etc. have to be propagated across the copies, which means an automated process for each new backups being integrated into the changes to the backups. But this is pretty much standard stuff. For the structure as identified, I do this by using rsync over ssh between repositories when in the maintenance mode, then remount the filesystems read only for use.

Other 'security' considerations

It is noteworthy that I have intentionally ignored lots of other 'security' considerations. For example, we have ignored:

- **Integrity:** Other than the backup scheme itself being oriented toward integrity of the history of content and the use of MD5 checksums and length as redundant checks on non-alteration of content, the use of append-only or read-only file system and FUSE components, there are no extraordinary integrity components provided.
- **Availability:** The availability of content, in the sense of redundant copies of content, is inherently reduced by consolidation, but can be compensated for by redundant copies of the backups. The big availability win comes from the reduced size and and increased accessibility of backups, which through consolidation makes the reality of getting to all of that old data far easier and more likely to actually happen. Similarly, the ability to trace provenance issues more easily and create subsets of the backup over time, location, etc. are improvements in availability from a practical standpoint.
- **Confidentiality:** Nothing in this approach inherently improves confidentiality. There is the advantage and disadvantage of keeping all of your eggs in one basket. Nothing like encryption has been mentioned here, and for authoritative content in physically secured facilities, this is appropriate.
- **Use control:** No use control mechanism has been offered or supported.
- **Accountability:** Nothing in this approach enhances accountability other than the improvement in accessibility.
- **Transparency:** Nothing inherently improves or alters transparency in this approach, however, the linkage of content across multiple instances and systems makes transparency for forensic and other investigative purposes far more accessible.
- **Custody:** Chain of custody information is not preserved in the approach, although improved accessibility of metadata can be useful in making this information accessible.

Clearly, there are other security issues associated with backups, and of course that is beyond the scope of this particular article. But on the other hand, in seeking some way to find relevant parameters to security issues associated with this approach to backups as opposed to others, I do not see any obvious problems created, while I do see improvements through the approach and process.

What to do with all those old backups

My backups are typically stored in ‘spinning reserve’ (old hard disks no longer used for their original purpose or taken from aged out systems after transfer of retained content to other systems). I have a few USB multi-port powered connectors that handle up to 10 USB devices, and since disks typically only spin when used, they are plugged in and available for use in 10-15 seconds as/when needed. Over time, I have accumulated perhaps 100TBytes of these disks, typically in sizes up to 4Tbytes and as small as 1 Tbyte. Much older ones are disposed of in an appropriate manner or stuck in devices awaiting disposal.

As it turns out, the file structure identified for this approach, as implemented for now, uses symbolic links to attach to the directory structures with the content, while transaction logs suitable for database or other mechanisms of implementing a FUSE file system are reasonably limited in size (4 Gigabytes for 26 million pathnames, or about 153 bytes per pathname). As a result:

- The files used to access the content and associated it with path names and structures, metadata, etc. are small enough to easily fit on a trivially small disk in today’s terms.
- The file structure does not need to be on a single disk or fused into a RAID array in order to be used as if it were on a single disk.

That last part is very nice for splitting backup content across multiple disks, or even over a network. Recall that there are 65,000 directories (256 at each of 2 levels) over which, for practical purposes, content is evenly spread. The MD5 checksum or similar approaches provides for this spreading of content files based on filename (MD5 checksum and size) while the likelihood of all the large files being in one portion of the file structure is a standard statistical analysis question that I will not answer here, but… I did do a “du s *” in the top level directory (containing 256 directories named 00 – ff) after the first 20 or so Tbytes of non-consolidated content and found that … See Appendix A for more details...

> Note that I did not do this the fast way by using the files with the metadata, but used the directory traversal version because, when tested, the FUSE file system was not yet implemented and I figured it was easier to just type the command (lots of disk and CPU time, but almost none of my time). Once the FUSE filesystem is implemented, this goes much faster because instead of directory traversal with lots and lots of disk access, there is a file that can be read once, parsed, and partial sums taken of one field. I actually tested this and it’s a few orders of magnitude faster of course, but rather than spend the time writing a small awk script, I chose to write this paragraph...

All you have to do to split the repository across as many disks as desired with symbolic links to each disk’s content area for relevant sets of content (00-b3 on one disk, b4-d8 on another, etc., sized based on available disk), and everything runs as if it were on a single disk. And that’s what I did with the old spare disks. I plugged a few into each of a few backup systems I keep around, and lo and behold, redundant backups. I use rsync to keep them up to date.

Conclusions

Backups with proper restoration and management over time are hard to do at all, and harder to do well for desired purposes. While this is not the last solution that will ever be needed, it is relatively simple to do, inexpensive to operate over time, and once consolidated, makes the whole incremental process for easier to track and manage. At least for me…

Appendix A: Details

Doing a du -s * from the top level Content directory, we found the following space usage:

Size	Directory	Size	Directory	Size	Directory	Size	Directory
1260932	00	3851376	01	622104	02	790540	03
786248	04	926580	05	644080	06	595544	07
597432	08	1763628	09	584852	0a	661248	0b
633124	0c	816432	0d	603500	0e	755512	0f
808496	10	741528	11	599292	12	930836	13
659092	14	2499936	15	3711328	16	599748	17
783512	18	852060	19	646640	1a	978068	1b
465764	1c	1111588	1d	738988	1e	1107672	1f
1051708	20	768720	21	1016720	22	682668	23
689356	24	879876	25	1048424	26	679488	27
803296	28	994768	29	1698812	2a	863216	2b
2746256	2c	748644	2d	953516	2e	776556	2f
1547328	30	686476	31	591520	32	3742636	33
4241772	34	2305908	35	938908	36	1755356	37
802500	38	812956	39	685604	3a	574524	3b
745356	3c	629668	3d	822420	3e	1479744	3f
1618156	40	989808	41	1114504	42	806864	43
1301144	44	1783488	45	1552344	46	654052	47
1648072	48	3136020	49	766140	4a	962884	4b
1642356	4c	727088	4d	954184	4e	683032	4f
1023020	50	865184	51	667912	52	710468	53
948312	54	3867760	55	802300	56	1720652	57
992116	58	1449504	59	779588	5a	601312	5b
724904	5c	1185928	5d	30461912*	5e	1146908	5f
690936	60	714068	61	810192	62	749040	63
1243344	64	868208	65	1264984	66	885512	67
738128	68	887436	69	816920	6a	703124	6b
883612	6c	595444	6d	871288	6e	607036	6f
658568	70	657688	71	1199572	72	861492	73
774972	74	2724120	75	4498676	76	767396	77
1553608	78	547840	79	2001940	7a	821968	7b
893140	7c	761008	7d	1201012	7e	603664	7f
678040	80	892044	81	727136	82	1278160	83
663612	84	771688	85	2908076	86	1223260	87
907468	88	696216	89	547976	8a	877020	8b
1212668	8c	3783380	8d	812720	8e	551492	8f
565672	90	638048	91	1361300	92	880812	93
781460	94	614540	95	1276932	96	3226844	97
736128	98	846828	99	1073084	9a	968032	9b
637680	9c	850704	9d	709288	9e	1371396	9f
762436	a0	2094736	a1	861452	a2	610140	a3
2294480	a4	540888	a5	671060	a6	1058748	a7
1320112	a8	821380	a9	573372	aa	810896	ab
740824	ac	673324	ad	672496	ae	1287552	af
1065288	b0	835728	b1	1174092	b2	634784	b3
741928	b4	1474560	b5	902188	b6	743016	b7
544408	b8	591424	b9	3979268	ba	1254832	bb
1024616	bc	1689956	bd	575376	be	694108	bf
913844	c0	2093432	c1	775372	c2	784268	c3
4952280	c4	841092	c5	1715824	c6	960468	c7
979340	c8	1060416	c9	738336	ca	1235240	cb
773776	cc	2179512	cd	719744	ce	570648	cf
596220	d0	1917424	d1	1127176	d2	838596	d3
2879724	d4	3371204	d5	1011416	d6	712976	d7
776516	d8	865124	d9	750204	da	1341712	db
742764	dc	1020524	dd	1017004	de	612864	df
731192	e0	592996	e1	634484	e2	750344	e3
1121228	e4	802192	e5	862664	e6	722764	e7
823040	e8	692132	e9	1269112	ea	1143896	eb
595132	ec	924424	ed	668964	ee	952004	ef
940608	f0	706216	f1	796828	f2	1668220	f3
719932	f4	1061328	f5	698492	f6	597872	f7
1041904	f8	507464	f9	1277660	fa	459376	fb
477612	fc	1491968	fd	1300264	fe	2326272	ff

*The 5e directory used an order of magnitude more space usage (30,461,912 1K blocks) than any other directory. Upon examination, a hard drive image from a 2003 forensics case is 30,005,821,440 bytes out of the total 30.461 Gigabytes, essentially the entire content. That was turned into an empty file using the deletion process, and a new file created adding the extension DELETED containing notes on why and how it was deleted when by whom. Disk usage for 5e then became 1,159,352 1K blocks, close to the average size range.

The total size (in blocks) from this sample was 283,756,468 1K blocks (283 Gigabytes) / 256 for an average size of 1,108,423 1K blocks, or 1 Gigabyte per storage area (00-ff directories). The range of sizes went from 459,376 (half a Gigabyte) to 4,952,280 (4 Gigabytes), thus facilitating easy storage on even reasonably small modern disk drives. Of course if the size stays this small, it will all fit easily on a single 1Tbyte drive, consolidation bringing backups down in size by at least a factor of 40 for this example. And this does not include the post-consolidation phase (except for the single removal identified).

Appendix B: What are the odds?

One of the challenges for using the MD5 checksum alone for storing large numbers of files is the likelihood of a ‘hash hit’, a situation where more than one file has the same MD5 checksum (in this case). In this case, we have added the file length to the MD5 checksum name, explicitly because I know of at least one such hash hit in my backups intentionally created by prepending a known MD5 homing sequence (a sequence that returns the checksum to its initial state, thus producing an identical checksum but for a file of different length) to a file. But the more general issue of hash hits by accident has to be considered, especially for backups with a large number of files.

First the simple problem. On average, how many files of the same length have identical MD5 checksums? Since an MD5 checksum is 64 bytes long, at best, every different file of 64 bytes produces a different MD5 checksum, thus consuming the entire space of all possible MD5 checksums. Of course this applies to any any cryptographic (or other) hash function. So every additional byte multiplies the number of instances of any given hash value for all possible files of that length by a factor of 256 (the number of values for a single byte). For for a given file length N (in bits) we know that the average number of identical MD5 checksum values for that length is 2^(N-512). For a 512 byte file that comes to:

> 1340780792994259709957402499820584612747936582059239337772356144372176403007354697680187429816690342769003185818648605085375388281194656994643364900608409 6

instances of each possible MD5 checksum (on average). But that is for all possible files of that length. To figure out the actual odds of 2 different real files of identical size having identical MD5 checksums, we need to use the Birthday problem solution as an approximation.

The birthday problem is the problem of calculating the odds that two people out of a group of N people will have the same birthday. We know that for 366 people, except for a leap year, it is guaranteed that at least 2 have the identical birthday. But what if we had only 10 or 20 people in the room? The answer is that only 23 individuals are required to reach a 50% probability of a shared birthday, but the solution in general is approximated by $1-e^{(-n^2/2d)}$ where n is the number of ‘people’ (in our case files) and d is the number of days (in our case checksum values). For 500 files of the same length, this comes to odds of a shared MD5 sum of about 1 in 10^{72} and the only question left to ask is how many files of each length are in the backup and the acceptable odds of an error caused by a hash hit of files of the same length.

Because filenames in our repository end in – followed by the length of the file in bytes, this can be relatively easily done by looking for repository content by filename ending and counting how many are found. For example, we did ‘find . -name "*-119" |wc’ at the root of the storage area and found 895 files of length 119. We found 1131 of length 120, and so forth. Of course this can readily be done from the table of contents directly and far more quickly. We took a bit of a sample and found that for a repository with just under 5 million files, we had:

Len	Quant	Len	Quant	Len	Quant	Len	Quant	Len	Quant	Len	Quant
200	685	500	486	1000	1181	10000	124	100000	4	1000000	0
201	2287	501	355	1001	909	10001	68	100001	2	1000001	0
202	700	502	402	1002	1014	10002	72	100002	4	1000002	0
203	498	503	339	1003	979	10003	83	100003	5	1000003	0
204	793	504	379	1004	1036	10004	76	100004	2	1000004	0
205	556	505	347	1005	1304	10005	64	100005	6	1000005	0
206	952	506	391	1006	1106	10006	71	100006	2	1000006	0
207	2993	507	418	1007	952	10007	80	100007	4	1000007	0
208	2469	508	427	1008	1090	10008	51	100008	3	1000008	0
209	23086	509	374	1009	973	10009	70	100009	1	1000009	0
210	549	510	400	1010	1011	10010	71	100010	0	1000010	0
211	477	511	355	1011	975	10011	81	100011	3	1000011	0
212	606	512	1709	1012	1097	10012	64	100012	2	1000012	0
213	505	513	524	1013	997	10013	63	100013	2	1000013	0
214	587	514	388	1014	1123	10014	71	100014	0	1000014	0
215	433	515	367	1015	1064	10015	67	100015	0	1000015	0
216	730	516	455	1016	1071	10016	73	100016	5	1000016	0
217	414	517	703	1017	1008	10017	73	100017	2	1000017	0
218	470	518	680	1018	1090	10018	78	100018	1	1000018	0
219	349	519	515	1019	1022	10019	82	100019	0	1000019	0
220	591	520	422	1020	1079	10020	77	100020	2	1000020	0

Obviously, as file lengths increase, we get fewer files of a given length. Why is that obvious? OK – it’s not. These are empirical results. Length 209 had a particularly large number, so it seemed like a good place to do a calculation. The odds of a hash hit in this case are about 2 in 10^{69}, which in my case is not alarming. Someone please check my numbers here…

Just in case you were wondering, if all of the 5 million files had the same length, the odds of a has hit would be about 1 in 10^{64}, which is still pretty small, and for a trillion files of the same length, the odds are about 4 in 10^{54}, so it seems pretty safe. But as I explained, there are known methods of producing the same hash value by adding to a file, so length helps a bit.

Appendix C: Some methods for disposition, search, and so forth

The filesystem I created uses the ‘stat’ command (linux) to produce values for times etc. and stores them in the format specified, in this case:

```
stat --printf '%W %X %Y %u %g %a %s
```

X, X, and Y are file creation, access, and modification times, in seconds since Epoch (in this case midnight, Jan 1, 1960). Then come user, group, access rights, and size in bytes. Of course you can store whatever you want, including inode information, the first characters of the file (for type identification in many case), and so forth. Given a set of this information, there are a range of things that can be easily done.

For example, disposing of all ‘files’ not accessed in the last 5 years, you can look for access times prior to the last 5 years (by converting the date and time 5 years ago to seconds since Epoch) and looking for all directory (TOC) entries with access values less than that converted time. All of those are easily removed from the TOC. But as you do this, you also collect all of the MD5-Length names from these entries, because the same content might exist in more recently accessed entries. Next you check the remaining TOC for any of those content files, and if no examples are found, you remove the content file. If examples are found, you do not remove the content file, because the content was accessed after the identified date. Even in this simplest implementation, this is linear time in the number of entries in the TOC (actual deletion may be linear time in the total size of files actually deleted in any of the partial repositories if you choose to overwrite the files with 0s before removing them from the file structure). Of course if you have redundant repositories, you will have to do the same TOC removals and file deletions (with optional overwrites) for all of the copies.

Obviously, you can delete files from date ranges in the same general approach, select out portions of a filesystem instance and do the same things, and so forth.

Redaction is another interesting case, as was described above. You can redact content from existing versions, but there may be different versions of redaction over time. The redaction explanation file identified above can track this, and using the same pathname for the content file as was originally used you can update historical content with the redaction, but new versions will have a new checksum and length, so it becomes worthwhile for some use cases to save an independent copy of each redacted version, linking each to the next with more redaction files, and so forth. But it’s important to remember that this is a backup system, and not a live filesystem intended for use by normal real-time applications. It does not have all the locking features of a normal file system in normal use. It is normally read-only except for maintenance periods when you would NOT want programs to access the content until the maintenance period ended.

Appendix D: Reconstruction

It turns out reconstruction of file systems is pretty easy in the simplest cases. Because the table of directories (TOD) is retained, any given file structure can be reproduced by simply running through the TOD and creating symbolic links to repository files in place of files found at the desired time for reconstruction, a representation of what was present in that time frame can be produced. Of course there are various issues, such as whether a given file or directory was actually present at the identified time.

Because backups are typically performed at fixed time (over limited time spans), and files may be added and/or removed between these time spans, the information is not present to perfectly accurately reflect what was present. Unless there is a record of filesystem transactions that can be used to augment the backup mechanism, reconstruction will leave alternatives for any selected time. Also note that the directory times and file times can get strange in reconstruction because, for example, we may have a directory change time before a file change time in that directory.

Also note that a backup typically runs over a time period and there may be changes to the filesystem during this period. This a file added or removed during backup may not be reflected as present or missing in the backup itself. While an opened file that is deleted during the backup process will typically be kept until the file is no longer in use, things like file system failures and removal of media during operations produce errors in backups that these approaches cannot mitigate. The directory entry may have been backed up unchanged for the last year, and in the time between the directory recording in the back and the backup of the files in that directory, a file might have changed, producing a date that is a year later than the directory entry is recorded as being changed. Similarly, files may be added or removed during this time frame.

An interesting reconstruction approach is to offer a previous and subsequent version of the reconstruction. To get a sense of this, if I have a backup with the most recent date and time stamp of Jan 21, 1998 and the next backup has a latest time stamp of Jan 30, 1998, and I want to know what the filesystem looked like between those times (because what I asked for stems from a legal requirement, or I don't know the actual dates and times of the backups) I cannot produce a precise version. But I can produce a definitive before and after, and they become alternatives to be used depending on the desire of the user. In doing this, I can also produce a differential, essentially saying that the things in (or not in) the "before" version and not in (or in) the "after" version are the things that changed over the period from before to after.

Appendix E: Provenance

Provenance information is another interesting application. Because the content is stored based on content rather than based on names, once I identify content a content file, I can find the earliest instance of it by simply selecting the TOC entry with the earliest modification date.

But that does not complete the picture. How it came to be is trickier than just the bits and bytes. For example, suppose the item of interest is a version of a document. We might reasonably want to know about previous version of the document not yet identified. At the level of a backup system, we can only do so much of course. The content of files might have to be inspected and analyzed in order to find actual traces of prior versions and related internally stored metadata from things like Word documents that are composed of multiple other documents.

But there are various methods that these backups lend themselves to and that are worthy of mention. Two lines of approach are tracking pathnames and measuring similarity of content.

- Pathnames for the same files change over time are a useful way to track provenance. The identical pathname for various content reasonably implies provenance, and you can then go back in time based on timestamps.

- Start with an instance, use the content name (i.e., MD5-Length) to find all instances and dates from the TOC.
- For each instance you get a pathname.
- Then look for other dates with the identical pathname to find other versions of content, and recurse.
- This will give you a list of pathnames, content, and date and time stamps of related files.
- You can then construct a provenance POset of all the identified versions.
- You can also do things like identify time frames in which content was duplicated or moved within a filesystem or between filesystems.
- From the linkages of files, you can also try to link directories containing them to find the broader history of the context and do multiple reconstructions from different time frames. This helps deal with different versions of larger sets of things than files.

- Similarity measures are another approach to finding provenance. In this approach, we use methods like Bloom filters, neural networks, or other methods to look for similarities between files.
  - From a backup, where we only have one copy of each content instance, we generate a result similar to the MD5 checksum in the sense that it runs through the content and produces a vector of values, network structure, or other outcome of analysis for each content file.
  - We then use the similarity detection mechanism to seek similarities with an identified instance we are seeking provenance for, and run it across all of the values or structures. This yields, typically, a ranked similarity list with the most similar content files associated with high metrics (e.g., 97.23%) and less similar items with lower values.
  - Using a Bloom filter as en example,
    - We can produce vectors for each file relatively quickly, leaving the results in a database, linear file, or whatever structure is desired. Thus we have the feature vector for all of the files readily available for multiple comparisons over time.
    - We can then run comparisons of any one feature vector against all other vectors if desired.
    - In most cases, similarity over time for individual files changed by user processes appear in files of similar size. For example, every edit of a text file is likely to produce a similar sized file. By limiting the comparison to similarly sized files, the search for similarity is, presumably, far faster, but there is a tradeoff between time consumed (more or less) and missed similarity (less and more).
- A different variation on this theme uses file differences (e..g, the diff program) to detect differentials between content files. However, this is a pairwise comparison only, so going through an entire backup would take N^2 differentials for all file pairs.

- The resulting metric would likely be something like number of lines changed or the length of the diff output.
- However, for follow-up from files identified by the pathname or similarity metrics, using differentials can be an effective way of tracking provenance for various applications of the backup system.

For provenance supporting repositories like GIT and similar development tools, there may be specific advantages in analysis of provenance, but these are beyond the scope of the discussion of backups here.

Appendix F: Malicious or undesired or controlled (i.e., bad) content

Another major problem today stems from the infiltration of viruses, malware, or other undesired, unlicensed, malicious, or unauthorized content into the backup system. Once present, it can end up being restored to a live system where it can cause further damage, even after previously being eliminated. It also consumes space and potentially continues to violate contractual, regulatory, legal, management expressed desire, or other constraints. I will call this ‘bad’ content to ease of use.

Ideally, bad content never enters backups and never gets restored from them. But since that is simply not the case, the question remains of what to do about it. The solution evident from history is to detect it and delete it in the live systems. But since we don’t always know what is or was bad when a backup is done, this is problematic. Here are some advantages of the backup system discussed here in that regard.

- In entering files into the backups, they can be checked using whatever technology is available today. In my case, the initial creation of the consolidated backup provides that opportunity, but I did not take advantage of it for a number of reasons.
  - One of those reasons is that, as a malware and virus research, I have substantial collections of historical viruses that would likely be detected by such a scheme and then removed from my backups, which I do not actually want.
  - Another reason is that these files come from many different systems, including forensic extractions from 3rd party systems, and in order to check for those systems I would need either a universal approach not commercially available or to do a complex reconstruction of those systems to use the available detection technologies from relevant time frames.

  But checking alone does not fully address the issues, such as false positives and false negatives and what is to be retained or removed.
- Suppose I could reasonably reliably detect bad files. Then what might I do with these?
  - There are plenty of options available historically, including effectively:
    - **Quarantine:** In this approach, we create a separate collection of content that is quarantined. Files can be moved to and from the quarantined version or area of the backups and when looking up content we can check both the quarantine(s) and non-quarantined repositories. This effectively retains the content but can be used to limit or eliminate its restoration except under management control. This is interesting for many reasons, including investigation, tracking of spread, etc.

    - **Deletion:** The problem with simply deleting such content is that multiple instances then have to be checked again. If you take the deletion approach from above, you can create zero-length files and mark them as deleted in the repository. Then you will detect the bad file when you do the checksum and not bother to store it. The reason for deletion can be identified as detection ad bad with any desired details at very little overhead.
    - **Marking:** A variation on deletion is using a marking scheme to identify properties associated with repository files. This could also be useful for other purposes, like classification of content by type as part of inspection to reduce effort in using the backup, or marking associated with classification levels, compartments, and similar such markings used in separation mechanisms. Marking can also be used for controlled content.
    - **Cleaning:** Some folks try to clean infected content by removing the bad part. If you can do it, that's fine, but the result will be different content, that should be treated perhaps similarly to redaction. In essence, we keep track of the original content checksum and size and refer to redacted versions so that redundant copies can not create redundant files in the backups.
    - **Retention holds:** For files that are being retained under a legal hold or for some other purpose, the disposition process can be implemented by moving to alternative repositories or by a marking process. This follows the same methods identified earlier.
- **Background checks:** In this case we do not mean checking the background of people. We mean using background processes in the backup repository systems to check backed up files. In essence, any check that can be done at entry or exit can also be done in the background, adjusting the backups to adapt to new understandings and methods for analysis. This addresses removal, reclassification, entry and exit from quarantine, and all of the other things we can do, once the backups are in place.
- **Restoration checks:** Checks not done on entry or in the background can also be done on restoration. This can be done in real-time to reflect the most up-to-date information and analysis techniques, and then back-applied to the repository for future retrievals as background checks, if so desired.
    - In many cases, restoration may not be complete for good reasons. For example, a backup restoration in a legal matter may be desired only including specific classes of content, from specific places, associated with specific user identities, or containing specific content. Restoration checks may be used for this purpose, thus only allowing desired content to pass.
    - The same methods used for prevention of exfiltration from live systems can be used for backup repositories and their retrieval.

It seems clear that this approach to backups can fairly easily be used for a wide range of purposes associated with so-called bad content while dramatically reducing storage space, analysis time, and retrieval time.

Final comments

As the end of year approaches, the numbers are piling up. I have now processed well over 500 million inodes covering something like 60 Tbytes of originating backup content, which after consolidation remains under 5 Tbytes of actual data including each version of each item changed over a 30+ year period. I am still going through old hard drives of computers that I was holding off on disposing of until I could recover or destroy their content, but there is no rush on adding these.

The process of creating a FUSE filesystem is underway but delayed – pending.

From my forensics work I had a fair number of disk images, and ultimately even made some of my own backups by making disk images. Not the nest idea for back and recover. So I decided, for the images with my content, to extract the content and abandon the disk images themselves. For this I used ‘partx’ to setup the partitions on the disk for loopback interface mounting, mounted them, copied the content of the file systems (I used tar to avoid copying things like deices that might actually go and copy existing devices rather than what might have been mounted on that system at that time), and after extraction, deleted the disk image, then completed the backup conversion process on the resulting file system.

More updates will be added as the project completes...

