Chapter Five

`/cloud-history`: Searchable Backup

Of course, no story about storage is complete without backup. Persistent storage maintains all long term information for individuals and businesses. Business data has a direct relationship with revenue for many businesses, and individuals are increasingly storing important memories in digital media formats. Storage is, therefore, subjected to immense amounts of protection. For example, common backup strategies include keeping three copies of data in at least two formats, with one copy off-site in accordance to the 3-2-1 rule [100]. Intriguingly, as in the beginning of this dissertation with virtualized storage, we find that the backup storage landscape is also changing. Backup storage systems no longer have the slow, high latency properties of technologies traditionally used for archiving data such as tape. Modern backup storage systems are based on faster storage technologies such as arrays of magnetic disks with access times on the order of milliseconds to seconds, instead of the slower minutes to hours for retrieving and reading tapes. Public clouds reflect this several orders of magnitude difference in access times across archival storage offerings. Amazon's Glacier [30], priced at 1 cent per gigabyte month, takes 3-5 hours to begin retrieving data. Google's Nearline [81], also priced at 1 cent per gigabyte month, takes 3 *seconds*—three orders of magnitude faster. Enterprise backup storage systems such as Sepaton [107] are also based on magnetic disk technology and have retrieval times of seconds to minutes. Three orders of magnitude faster access to archived files opens up a new opportunity for rich queries over history previously trapped within slow to access and infeasible to query backups.

`/cloud-history` is a dual exploration into architecting backup with DS-VMI and also optimizations reducing the time to index archived files. This chapter answers two questions: if we assume a cloud built with DS-VMI, *how would we architect backup differently*? And, given modern fast access backup storage systems, *how can we efficiently index backup data*? In answering the first question, we find that simply storing the file-level update streams provided by DS-VMI gives a data structure and format which exactly matches the needs of backup. The log-structure maximizes sequential throughput, and enables retrieval of previous versions by reverting or replaying events in the log. The generality of the second question makes it hard: we do not limit the type of index, nor the type of data being indexed other than assuming it resides within files. Thus, we explore application-agnostic methods of speeding up the time to index over general files. This second question, and indeed our solution, is orthogonal to the capture mechanism—DS-VMI—although we imagined solving it in the context of DS-VMI.

This chapter is organized as follows. Section 5.1 describes at a high level the benefits of using DS-VMI to capture historic state. However, the optimizations and results of this chapter are independent of the capture mechanism. Thus, the lessons learned here are applicable to a class of backup systems which feature fast time to retrieval. Section 5.2 provides a first of its kind analysis of backup data with a look at whole-file deduplication not for saving space, but for saving computation time. Section 5.5.2 describes our technique of creating versions of files without system call feedback. We resort to a heuristic-based approach, as we can not decide when a userspace process "saves" a file. Section 5.3 describes the amount of file-level deduplication we expect, how we implement it within the DS-VMI framework, and its effect on indexing workloads. Section 5.5 deals with storing file logs and versions efficiently on disk for maintaining a high ingest rate, and a quick retrieval time. Section 5.6 describes how we use another FUSE driver to retrieve historic versions stored on-disk within file-level log streams. Section 5.7 describes securing the search mechanism, and how secure search is implemented.

5.1 Transforming Live State into History

In this section we provide an overview of the properties we want in transforming the cloud's virtual disk state into a format suitable for archiving and indexing. These properties guide two key architectural decisions: how do we capture historic state, and how do we store that state for future indexing?

5.1.1 Desired Properties

We focus in this chapter on three important properties:

1. Capture complete, tamper-free history
2. Scale across different OS's and applications
3. Support semantically meaningful, efficient indexing

Preserving integrity means tamper-proof capture of the complete history of virtual storage. It implies independence from guest faults and compromises. Thus, any solution fulfilling this property needs isolation from the guest environment. In addition, the desired solution must generalize across different guest environments without requiring guest cooperation. Finally, the desired solution must provide efficient access to a semantically meaningful version of history. Otherwise, indexing becomes inefficient, and in the worst case intractable.

Given our desired properties, we explore applying DS-VMI as the state capture mechanism most well-suited to our problem. DS-VMI provides isolation by leveraging the strong isolation between a VM and its hypervisor. DS-VMI provides generality by capturing state at the level of virtual disk writes without any guest support. Yet, DS-VMI is not limited to the coarse-granularity of the low-level writes it captures. By intelligently tracking file system metadata, DS-VMI maps each write into its semantic, file-level interpretation. This makes DS-VMI ideally suited to capture historic state for `/cloud-history`.

Of course, either snapshotting or an in-guest agent, such as a versioning file system, could be used as a state capturing mechanism for `/cloud-history`. However, snapshotting requires additional processing to map each snapshot into its file-level interpretation, and does not capture a complete version of history. Versioning file systems, or any form of guest support, force constraints onto the guest environment and are vulnerable to faults affecting the guest. We ruled out both snapshotting and versioning file systems for these reasons, but recognize that with minor changes to our architecture below, they could replace DS-VMI as a capture mechanism.

5.2 Learning from History: a Backup Case Study

Is the third property discussed in Section 5.1.1 feasible in real world backup systems? Specifically, is expecting scalability of file-level indexing a valid assumption? Is there any way of increasing scalability, should it initially prove intractable, while maintaining the generality of indexing? And are backup systems capable of quick enough retrieval of data to make indexing and searching them a reasonable expectation? As we discussed in the introduction to this chapter, Google Nearline is an example cloud-based storage system designed for backup, with high bandwidth and low-latency data access which undeniably makes the answer to this question yes. Today, Google Nearline promises time-to-first byte between 2-5 seconds for a bucket, which is typically a set of files. The rest of this section deals with a study done over a large corpus of backup data in a quest to answer the first two questions.

5.2.1 Description of Dataset

In order to answer these questions and many more, we studied a large corpus of backup data from a production backup system. The backups consist of approximately 1 year's worth of backups, over 58 unique systems. Most these 58 systems are servers running a variant of Linux. The backup system, deltaic [13], automatically ages the backups. Thus, the first few crawled backups are monthly, then weekly, and finally daily. Not all of the 58 systems were included in backups for the entire time period. This is why the total number of system snapshots is less than the number of systems multiplied by the number of backups. In addition, some systems occasionally fail to backup further reducing the overall number of system snapshots. Our backups and system snapshots showed significant variance in the number of files per snapshot, and average growth in files per backup.

Statistic	Value
Number of Systems:	58
Number of System Snapshots:	3268
Number of Discrete Backups:	69
Time Period Studied:	1 year
Total Files:	1.676 billion (14 million deduplicated)
Total Bytes:	146 TiB (4 TiB deduplicated)
Average Number of Files per Snapshot:	512,898 (2,496,899)
Average File Size:	98 KiB (9 MiB)
Average Backup Growth in Files:	-10,826 (245,903)
Average Growth in Bytes:	681 MiB (24 GiB)

Table 5.1: This table shows the details of our study of a research backup system used in production support of a research group at CMU. Parenthetical values are standard deviations, unless otherwise noted.

We speculate that this variance comes from a single system with a very large amount of small files, which was eventually taken offline, thereby removing it from backups.

We anonymized the data by only storing HMAC's of the pathnames in our database. We do not maintain the private key used by our HMAC function. We are compatible with `rsync`'s notion of similarity, which was the primary tool used for capturing file-level state in deltaic. This means that if we have already recorded a certain HMAC and its modification time has not been modified, we did not count it as a new unique file. Preliminary statistics and a summary of our collected database is shown in Table 5.1. Most of the numbers agree with prior backup studies [71]. However, we observed negative average growth in the total number of files snapshotted due to an outlier backup which dropped over 1.7 million files at once. Excluding this outlier gives an average backup growth in files of 14,594 (standard deviation of 125,602).

Table 5.1 shows a two orders of magnitude drop in the total number of files vs the number of deduplicated files. As we will see in the next section, eliminating file-level duplicates has an immense impact on the time to index for real-world indexing applications. In addition, we note a similar two orders of magnitude reduction in the number of bytes. Although research on backups shows an even greater space savings with variable-sized chunk deduplication, we find that the metric with highest impact on time to index is in the number of files—not the number of bytes. This is directly reflected in our experiments.

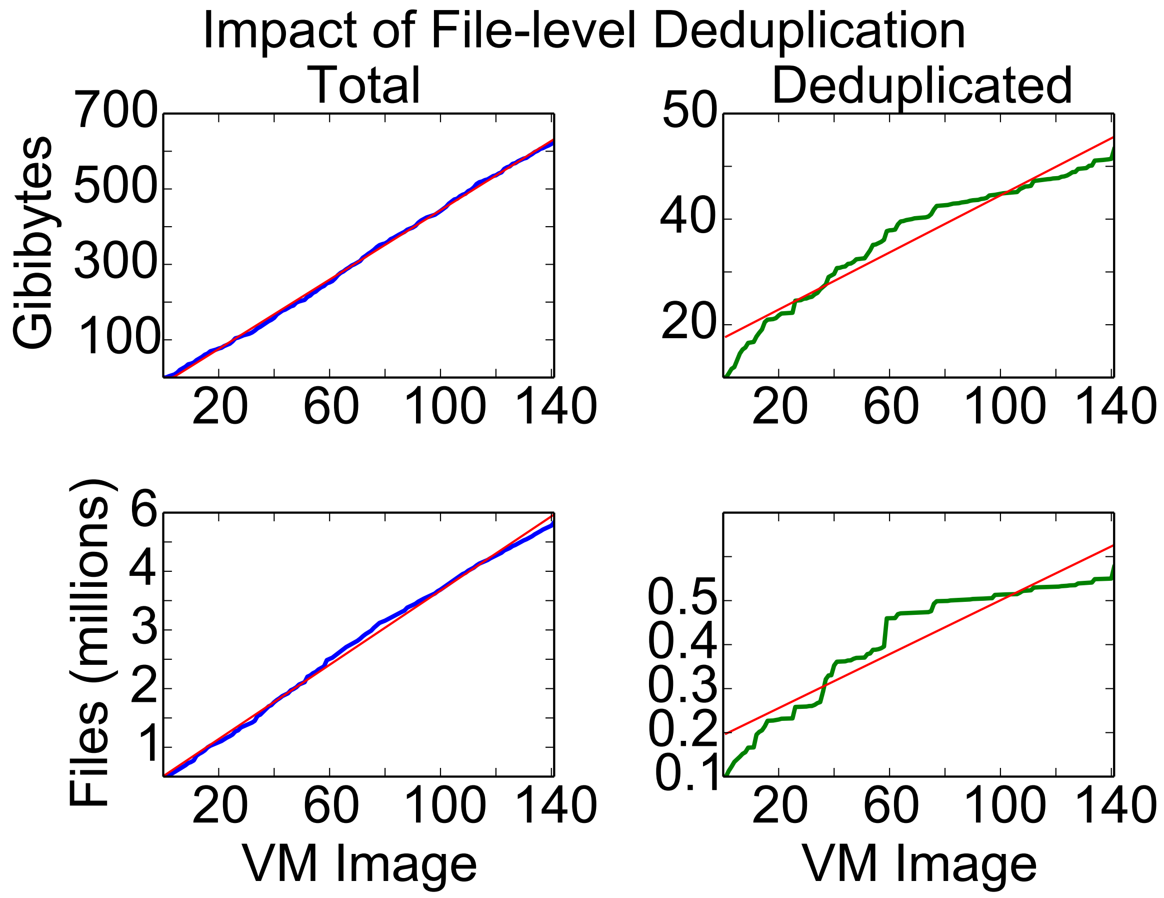


Figure 5.1: Deduplicating at a file-level leads to a 9.1x reduction in the number of files to index, and a 7.2x reduction in storage requirements (NC State, VCL Windows-based images). The raw bytes and files appear to grow linearly. The additional unique bytes and files appear to grow sub-linearly, and possibly logarithmically. Linear fits are shown as the red lines.

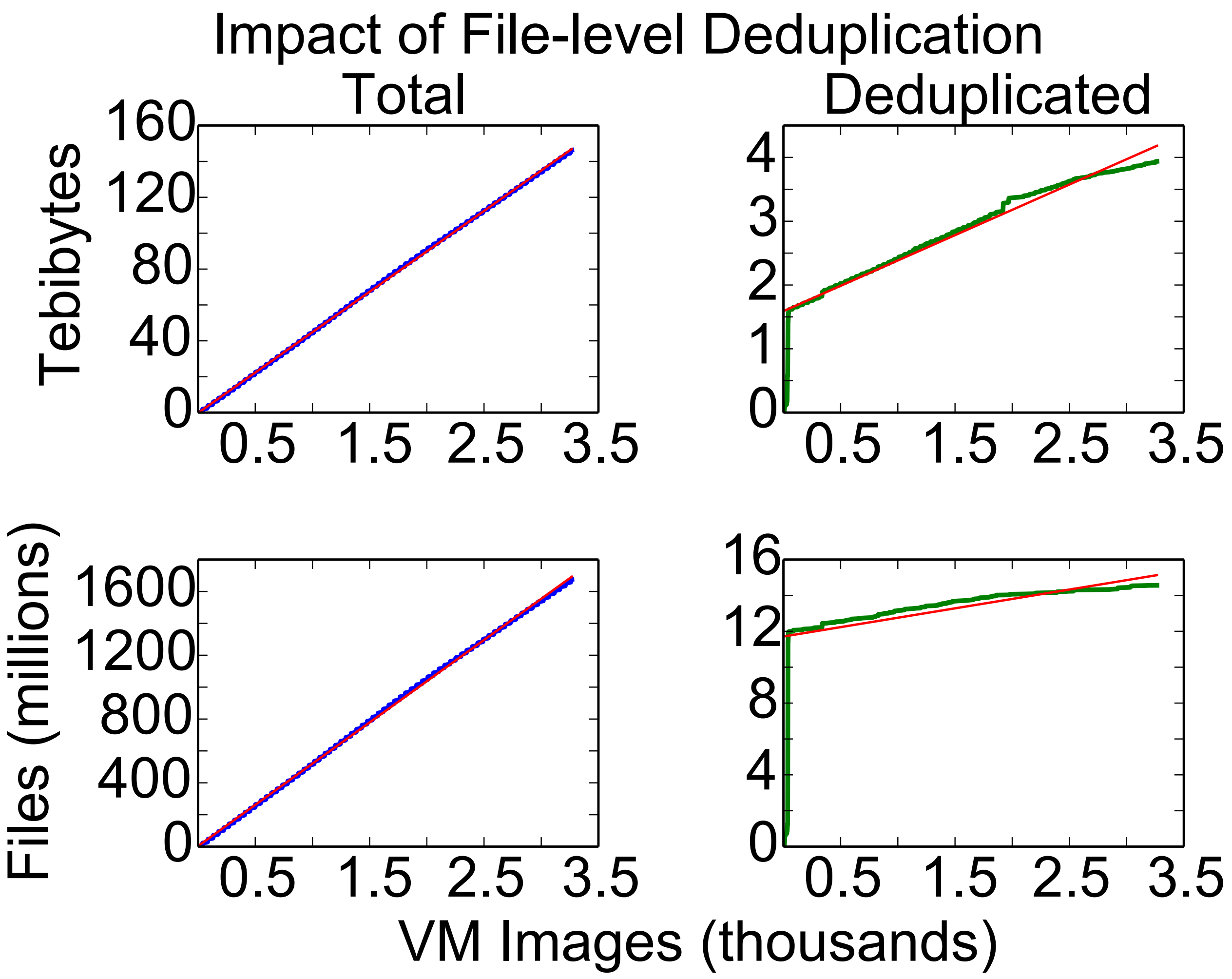


Figure 5.2: Deduplicating at a file-level is even more effective when applied to backups of systems. The unique file curve flat lines as more and more snapshots were taken of file systems by Deltaic. Linear fits are shown as the red lines.

5.2.2 Effect of Duplication

Although we had some early hints that file-level deduplication would greatly assist an indexing workload, we had no true validation that file-level deduplication works at scale until this backup study. Initially, we took a dataset of close to 140 virtual machine images from a cloud at NC State and investigated deduplicating them at a file-level. The results are shown in Figure 5.1. We observed a very promising 9.1x single order of magnitude drop in the number of unique files needing indexing, and 7.2x drop in the amount of space needed to store these images. These VM images had significant duplication between them because they were all based on Windows OS with only application-level customizations.

Although proving the point that significant deduplication exists across many VM images, we really want to know if it will be tractable to search at a file-level the snapshots of virtual storage. Figure 5.2, shows the results from an almost one-year study of the Deltaic backup system. We cut down the intractable 1.7 billion files stored in snapshotted file systems, to the more tractable 14 million unique files after applying file-level deduplication. In addition, there are significant storage savings to be had from close to 160 TiB stored down to less than 4 TiB stored. Both reductions represent two orders of magnitude reduction. As will be shown in later sections, the reduction in the number of objects to index drastically reduces the time it takes to re-index for various indexing workloads. Indexing workloads appear CPU-bound, not disk-bound. Thus, traditional deduplication at a block-level had very little impact on the overall time to index. This is an important point because it demonstrates the immense value of file-level deduplication for indexing workloads. Classic deduplication would need a second index over files not just blocks to speedup file-level indexing workloads. At the same time, classic deduplication offers the best savings in terms of raw bytes used. In reality, there is no reason to choose one over the other because file-level deduplication easily layers over block-level deduplication technology operating at a lower abstraction level. In fact, using them in tandem seems like the best idea because we can pair the bytes saved with quick file-level indexing.

5.2.3 Analysis of Trends

Putting this all together, we see that over time the number of unique files grows with a very low slope. It may asymptotically approach a constant value given enough data and workloads with little file-level churn. If we ignore the ramp up part of the curve with the initial set of 40 snapshotted systems, our slope becomes negligible and we are essentially asymptotically approaching 14 million unique cloud-wide files. The amount of bytes added by this smaller set of files accounts for 3/4 of the unique bytes in the systems studied. This is an interesting trend, because it implies that the new, non-unique files are fairly large. Potentially they are large multimedia files such as video files, although we do not know due to the HMAC scrambling of path names. The trend without deduplication is precisely as expected—the number of files and bytes grows linear in the number of systems snapshotted. This is expected because backup workloads repetitively add the same data

Version	Size (MB)	Files Same	Bytes Same	Release Date	Days Stale
2.10.1	60	100%	100%	12/14/09	0
2.10.0	59	88%	60%	10/26/09	49
2.9.0	57	57%	30%	04/17/09	241
2.8.0	53	40%	25%	10/20/08	420
2.7.0	53	35%	22%	04/22/08	601

Table 5.2: R Source Tree Similarity

Version	Size (MB)	Files Same	Bytes Same	Release Date	Days Stale
1.4.1	47	100%	100%	01/15/10	0
1.4.0	47	99%	78%	12/08/09	38
1.3.4	48	98%	73%	12/01/09	45
1.3.3	47	88%	43%	06/14/09	215
1.3.2	46	82%	37%	04/22/09	268
1.2.9	33	20%	5%	02/17/09	332

Table 5.3: OpenMPI source tree similarity.

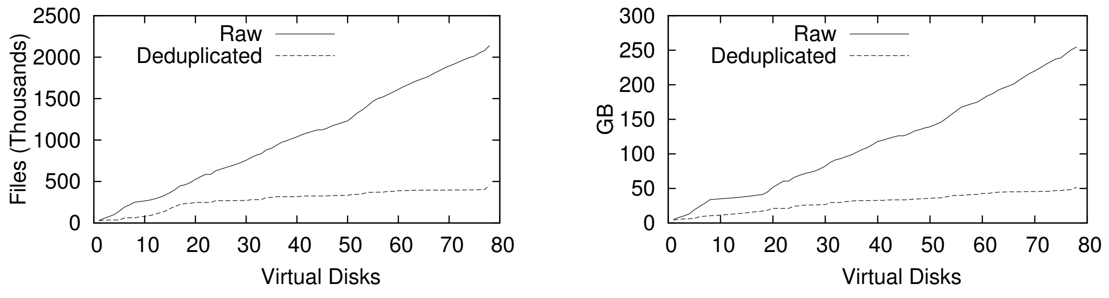
over and over again with each backup. Unless there is significant churn in the number of systems backed up, we expect generally linear growth.

Of pivotal importance to this chapter and the viability of indexing backup data is the trend of unique files. After the first backup, we see the vindicating trend that not many new files are encountered. There is a clear diminishing returns effect, which is exactly what we want for efficient indexing. After an initial index is constructed, smaller incremental updates are needed to keep it up to date. In addition creating new indexes is cheaper with such a large reduction in the file space. This discovery, along with the industry trend of ever cheaper storage with quick access times, makes a searchable backup system practical. As we will show, even a single node could index this cut down unique file space.

There is a step nature to all four graphs in Figure 5.2 which is due to a special system designated as an encrypted VM disk storage server which stores millions of small chunks for virtual disks in the form of small files. Such a system is not necessarily representative of cloud workloads, but it is a real example backup user.

Version	Size (MB)	Files Same	Bytes Same	Release Date	Days Stale
2.6.33.1	353	100%	100%	03/15/2010	0
2.6.33.0	353	99%	98%	02/24/2010	19
2.6.32.9	341	71%	49%	02/23/2010	20
2.6.31.12	326	56%	33%	01/18/2010	56
2.6.30.10	315	47%	27%	01/06/2010	68

Table 5.4: Linux Kernel Source Tree Similarity



(a) Size of virtual disk library in files. (b) Size of those files in bytes.

Figure 5.3: Effect of file-level deduplication on virtual disks in a virtual disk library.

5.3 Sources of Whole-file Duplication

In this section we develop the intuition for why large amounts of duplicate files reside inside and across monitored machines. Based on this developed intuition, we posit that rapid indexing of backup data is feasible by orders of magnitude reduction in the number of unique files to index. It is this intuition coupled with experimental indexing results that guide the addition of a unique file index to `/cloud-history`.

As an example of duplication that can occur within a single system, we studied three open source projects over large time scales. We expect source code to have a relatively high rate of file-level changes in comparison to installed system files, applications, or media files. On a day to day basis, developers touch many different files. But most system files, applications, and media files are written once and read many times. They only change during installation, upgrade, or removal.

Figure 5.2 shows the amount of duplicate files within the R open source project. The R project is a widely used statistical computing package. Yet, over a 2.5 year time period, over one-third of all files in the project remained identical. The files accounted for one-fifth of all on-disk bytes for the R source tree. This suggests that the frequency of change at the file-level is not high in general, even over long periods of time.

Figure 5.3 shows the number of files that are identical in different releases of OpenMPI. OpenMPI is a message passing library frequently used in the context of supercomputers and high performance computing. Even versions that are more than six months apart show substantial similarity: between versions 1.3.3 and 1.4.1, nearly 88% of the files are identical. These files account for nearly 43% of total source tree size in bytes. Other open source projects show similar results.

Figure 5.4 shows the amount of duplication within various Linux kernel source trees. The Linux kernel is one of the most active open source projects in the world, and is comprised of millions of lines of C code. Out of the three open source projects studied, the Linux kernel had the highest amount of file-level churn. However, over three months of releases almost half of the files remained identical, which accounted for 27% of the bytes.

We also studied real-world Windows XP file systems with a dataset from NCSU's VCL cloud. Figure 5.3 shows the impact of deduplication across systems at the file-level on 78 virtual disks from the VCL cloud. The number of files with distinct content grows much more slowly than the total number of files. Figure 5.3(a) shows less than 500 thousand distinct files out of two million total files in 78 virtual disks—almost all from a production cloud at NCSU. The storage capacity and I/O bandwidth savings from deduplication of these files is substantial: 50 GB rather than 250 GB, as shown in Figure 5.3(b). Although reproduced here for quick reference, a more complete picture continuing the trend lines for the NCSU cloud dataset is in Figure 5.1.

5.3.1 Impact of Whole-file Deduplication on Indexing Workloads

We studied four representative indexing workloads to understand the impact of whole-file level deduplication on their performance. The results are shown in Figure 5.4. `tar` [39] is a baseline program which has very little CPU usage, although it has to crawl all files as if it were an indexer. ClamAV [21] is an open source virus scanner which searches for viruses throughout all files. Recoll [94] is a full-text search tool for Linux and UNIX desktops. Apache Solr [116] is a full-text document search tool designed for high performance. The applications studied show an average speedup of 5x over this dataset. Cutting down the time to index is critical for making a flexible search system that is low-latency, efficient, and usable. Today, waiting for the retrieval of backup data may take hours. With `/cloud-history`, waiting for the answer to a new query might take hours. However, once indexed, future queries benefit from cached indexes with low-latency response times.

We assume user queries are deterministic: given the same object, they return the same result. A trivial optimization for minimizing user query time is result caching. Result caching helps queries by caching the results of previous queries. While it minimizes wasted CPU cycles, result caching only helps future queries. However, we can leverage the deterministic nature of queries to also optimize individual queries.

Many modern file systems compute hashes over the data in files to ensure data integrity [33, 99, 133]. Such hashes offer a backup system that indexes file systems a free opportunity to leverage pre-computed hashes for duplicate file detection. Even if a file system does not compute a hash

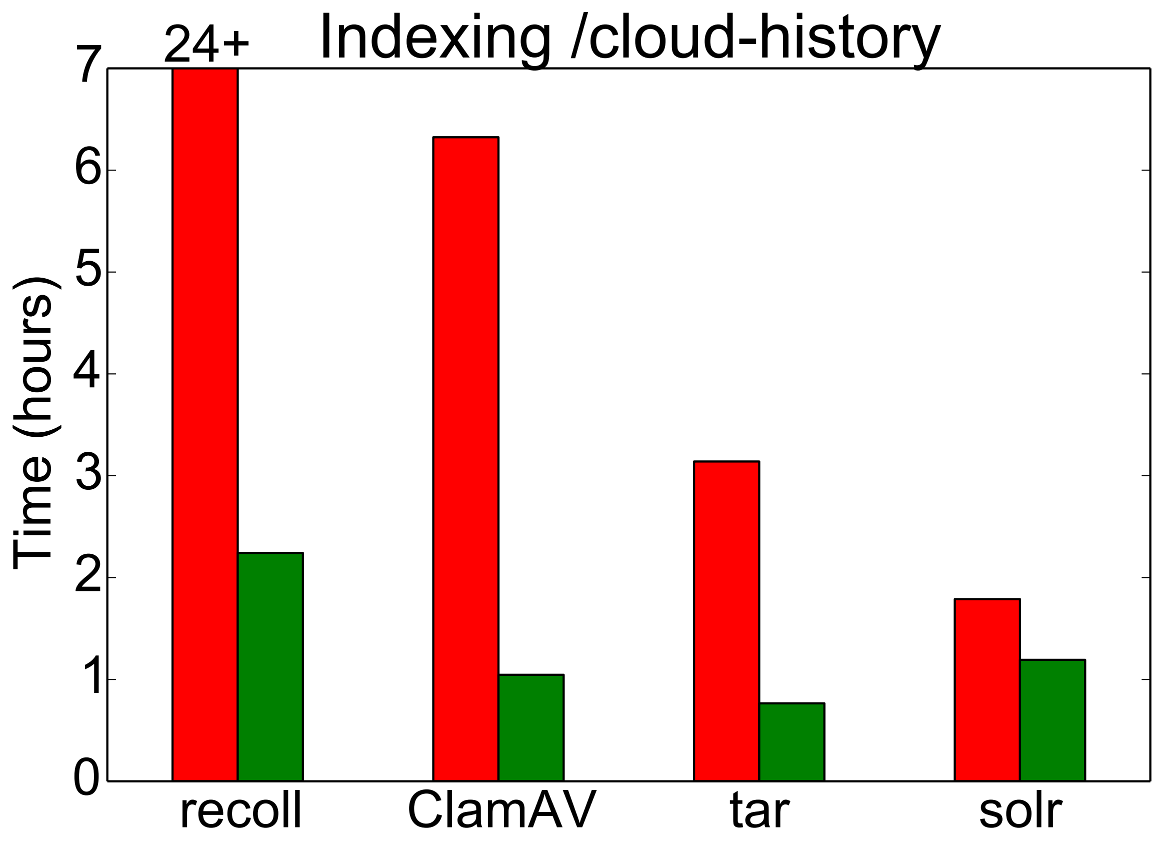


Figure 5.4: File-level deduplication not only saves space, it also saves immense amounts of computation. On average, the three workloads shown above experienced a 5x speedup by using just an application-agnostic unique file index. These experiments were carried out on a single node, on the NC State VCL Cloud image dataset.

over file data, hashing represents a fixed cost at the ingest of a backup system. There are many hashing algorithms and one with suitably high performance may be chosen to minimize impact on the ingest rate of a backup system. Thus, whether opportunistically obtained from a file system or computed at ingest, we can safely assume whole-file hashes are cheap.

Even single one-off queries, with no repeat or reuse of results in the future, benefit from duplicate tracking. They benefit by skipping duplicates resting on potentially slow backup medium, and eliminating the wasted cost of the query computation on duplicate objects. Of course, we implicitly assumed that backups are highly redundant. As shown in previous sections, this is a safe assumption because often backed up systems contain similar operating systems, libraries, and userspace applications which is what we found in the last section. For example, UNIX-like operating systems account for over 67% of all servers worldwide [125]. Microsoft Windows variants account for over 90% of all desktop and laptop computers [125]. Linux runs on 97% of all supercomputers [125]. Leveraging this similarity is an opportunity to optimize user query execution time.

5.4 Architecture of /cloud-history

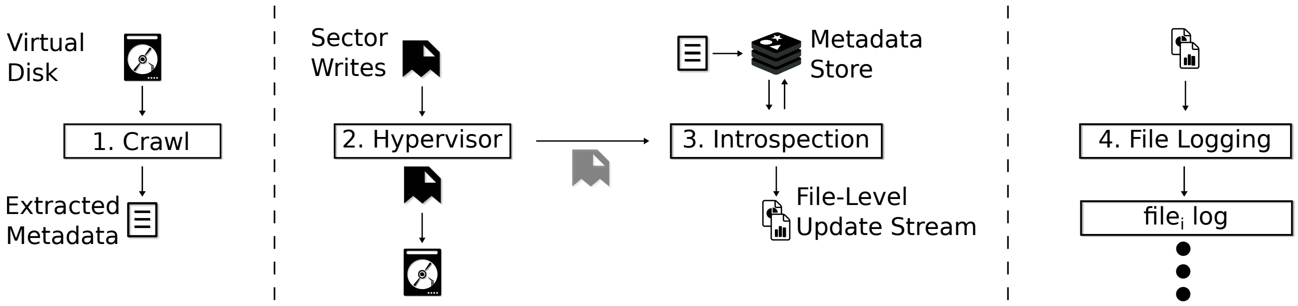


Figure 5.5: An overview of /cloud-history's architecture capturing writes via the hypervisor and translating them into a file-level update stream. The final stage stores the file-level update stream into per-file logs for indexing, garbage collecting, and application of retention policies.

Guided by the properties in Section 5.1.1, we chose DS-VMI as the technology to capture and map back into a semantic space virtual disk writes. In Figure 5.5, we review the stages of DS-VMI for virtual disks, and explain how /cloud-history stores logs of the output from DS-VMI which form the indexable history. We discuss the details of efficiently storing and indexing such logs in Section 5.5.4.

The first three stages remain the same, as they were described in Chapter 2. First, as shown in Figure 5.5, we scan the virtual disk (discussed in Section 2.3). This step extracts critical pieces of file-system metadata used at runtime to map virtual disk writes into their semantic, file-level meaning. This step can be performed either offline or online. Second, the hypervisor or network expose writes to the DS-VMI system (discussed in Section 2.4). This duplication may be turned on or off at will. The third step takes an arbitrary number of steps to resolve the semantic meaning of a write at the file level (discussed in Section 2.5). In the worst case, writes are not associated with any file system visible entity.

Once converted to a file-level update stream, traditional VMI then inspects the operation and emits the update to interested monitoring applications. In `/cloud-history`, we instead capture the stream of file-level updates into a centralized, persistent cloud store. We separate out the single large stream of file-level updates into per-file logs. Each log represents the state changes that occurred since the VM booted or was last snapshotted.

Finally, not shown in Figure 5.5, these streams are indexed to best serve the queries that users find important in `/cloud-history`. For example, a virus scanning indexer looks for traces of infections at any point of a file's life. This type of indexer inspects every update to every file. A document indexer, letting users quickly search through their documents such as Word or PowerPoint files, is only interested in files with certain extensions and their versions. A third type of indexer, a vulnerability scanner, is only interested in files containing binary, executable content. It indexes certain paths containing well-known binaries, and files with specific extensions. A vulnerability index, if kept up-to-date, is useful in answering questions such as, "which of our servers were vulnerable to Heartbleed [23]? and for how long were they vulnerable?" Such questions are important to answer because Heartbleed is exploitable without leaving a trace behind, and may lead to compromised customer data. Warning the right customers with the right dates requires answering such questions.

5.4.1 Consistently and Efficiently Naming Files without Coordination

Thus far, in this chapter, we have developed an intuition that lots of duplicate files exist both within and between systems, we experimentally confirmed this intuition, and we established the impact of whole-file deduplication on four representative indexing workloads. In short, whole-file deduplication is necessary for the scalability of indexing workloads running on top of `/cloud-history`. If it is such an important capability, how will we identify duplicate files across systems? At first glance, this seems like the perfect job for a hash function. However, modern cryptographic hash functions require re-reading all file data in order to update a hash unless they are appending writes. For potentially very large files, re-reading all of their bytes for every update is incredibly inefficient. Such a strategy would lead to the backup system slowing to a crawl as it tried to re-read large amounts of data.

Table 5.6, shows four competing methods for computing whole-file hashes. Hashing, H in the table, requires re-reading all bytes and results in the worst case $\mathrm{O}(N)$ run time. In addition, traditional hashing algorithms can not benefit from multiple processors. Merkle Trees, MT in the table, only require updating from a leaf to the root in a tree of hashes, and require a better, but sub-optimal $\mathrm{O}(\log_f N + 1)$. The special case single-level Merkle Tree, SLMT in the table, requires again a worst case of $\mathrm{O}(N)$, but without needing to re-read all bytes. Incremental hashing [12] takes an approach which provides $\mathrm{O}(1)$ updating of the whole-file hash for both random and sequential writes, and it can benefit from multiple processors. This is attractive as it is the most efficient known

Operation	H	MT	SLMT	IH
Update (S)	$\mathbf{O}(\mathbf{1})$	$\mathrm{O}(\log_f N + 1)$	$\mathrm{O}(N)$	$\mathbf{O}(\mathbf{1})$
Update (R)	$\mathrm{O}(N)$	$\mathrm{O}(\log_f N + 1)$	$\mathrm{O}(N)$	$\mathbf{O}(\mathbf{1})$
Update (B)	$\mathrm{O}(N)$	$\mathrm{O}\left(\frac{fN'-1}{P(f-1)} + \lceil\log_f N'\rceil\right)$	$\mathrm{O}\left(\frac{N'+1}{P} + N\right)$	$\mathrm{O}\left(\frac{N'}{P} + \lceil\log_2 N'\rceil\right)$
Space	$\mathbf{O}(\mathbf{1})$	$\mathrm{O}\left(\frac{fN-1}{f-1}\right)$	$\mathrm{O}(N+1)$	$\mathrm{O}(N+1)$

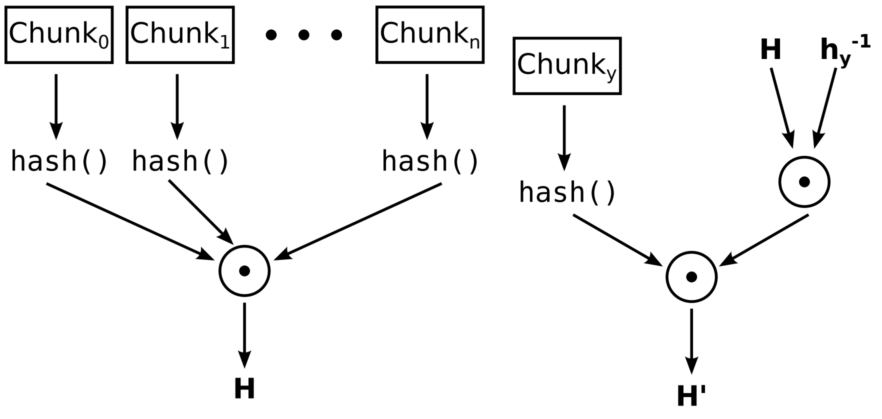
Figure 5.6: Running time of operations for three different hashing schemes. N is the number of blocks in a file, N' is the number of updated blocks in a batch update, f the fanout of the tree, and P is the number of processors. H stands for hashing, MT for Merkle Tree, SLMT for single-level Merkle Tree, and IH for Incremental Hashing [12]. For the updates, S stands for Sequential, for random, and B for batch.

hashing algorithm for updating a whole-file hash from a single write. In DS-VMI, we deal with single writes to a file all the time, and need a method of quickly computing a whole-file hash without re-reading all of the bytes of the file. Incremental hashing provides precisely this functionality.

In addition, incremental hashing is more space-efficient, requiring only $\mathrm{O}(N+1)$ storage. A Merkle tree requires $\mathrm{O}(\log_f N + 1)$ hashing operations to update a whole-file hash, and $\mathrm{O}\left(\frac{fN-1}{f-1}\right)$ space. However, for a large portion of file sizes experienced in practice the performance of Merkle trees may closely approximate that of incremental hashing. This is due to the decimation provided by having a large fanout at the bottom of the Merkle tree. This decimation keeps the Merkle tree short in height, requiring only a few extra hash operations over the simpler incremental hashing paradigm.

We envision extending DS-VMI with incremental hashing or Merkle trees for quick cross-VM whole-file deduplication. By batching updates across similar VMs, we expect whole-file deduplication to greatly reduce bandwidth requirements while providing continuous data protection to user-specified files. We need to perform a parameter sweep to determine the best default batching value across a mixture of write workload types.

Incremental hashing [12] is the only method which supports all of our unique hashing requirements. Incremental hashing provides a hash construction which supports random updates, is compact, requires no re-reading of data from the virtual disk, and offers collision resistance. As shown in Figure 5.7(a), incremental hashing works by splitting a file into n chunks, hashing each chunk using an "ideal" hash function, and combining the hashes using a group operation. As described in [12], a practical hash function could be from the SHA family, and an efficient group operator is modular addition or multiplication. Updating, shown in Figure 5.7(b), requires the inverse of the old chunk hash, the old hash, and the hash of the new block. Using DS-VMI we have the data needed as input to the hash function `hash`, and the other information must be stored as metadata. Modern file systems such as btrfs [99], and ZFS [133] already compute and store 256-bit hashes or checksums for every data block, thus external incremental hashing could have



(a) Incremental hashing of file chunks. (b) Updating an incremental hash.

Figure 5.7: The incremental hashing construction.

very low overhead. We propose using incremental hashing to implement file-level deduplication for `/cloud-history`, because it enables efficient re-computation of a whole-file hash on every write with no read requirements.

5.5 Storing and Indexing Historic State

`/cloud-history` collects file-level update streams via the DS-VMI mechanism introduced at the beginning of this dissertation. `/cloud-history` centrally demultiplexes these streams into per-file logs kept in a cloud secondary storage service. Each file log is separately garbage collected, versioned, and pruned. Users of `/cloud-history` may retrieve any version of any file across all of their virtual machine instances in the cloud. `/cloud-history` further maintains a unique file index over the files of individual tenants. This unique file index serves as the basis of efficient indexing. Building an index over backups, for example, which files harbor a critical vulnerability, is imperative for quick query response times.

In this section we describe our implementation of `/cloud-history`. This includes capturing writes with a hypervisor, and converting those writes into file-level update stream logs (Section 5.5.1). In addition, we discuss important operations over those logs such as versioning (Section 5.5.2), and garbage collection (Section 5.5.3). We finish the section by describing important optimizations making storage scalable, and indexing tractable: version deduplication to speedup indexing (Section 5.5.4), and supporting block-level optimizations for storage scalability (Section 5.5.5).

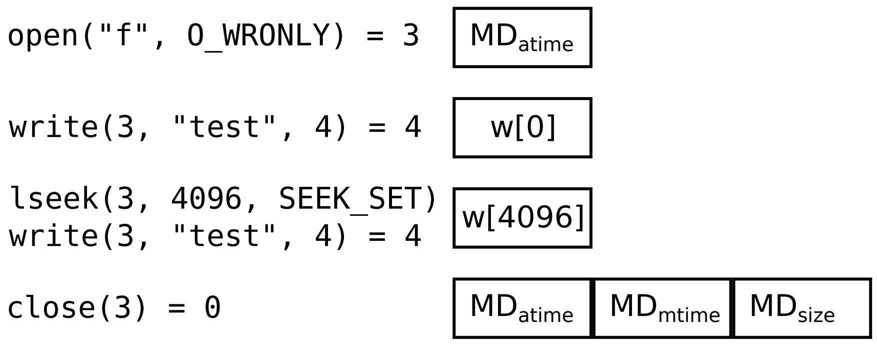


Figure 5.8: On the left-hand side are system calls occurring in userspace within a guest. These system calls cause block writes to a virtual disk. The corresponding block writes are introspected resulting in the high level events shown on the right-hand side of this figure.

5.5.1 Conversion to a File-level Update Stream

As shown in Figure 5.8, DS-VMI translates low-level operations initiated on behalf of guest actions into high-level events on files. `/cloud-history` captures these high-level events into a log file, and keeps one log per tracked file. Virtual machine introspection transforms block writes into a stream of semantically important file-level operations. Thus, a file system is represented by many such streams. This is contrary to the familiar single log structure of log-structured file systems. However, merging all of these streams into a single log is also possible if maximizing write bandwidth is desired. By operating on per-file streams, we have flexibility in this design space.

Log-structured files enable quick retrieval of the history of individual or sets of files. Traditional log-structured file systems must traverse much larger whole-file-system logs to reconstruct per-file historic state. In addition, log-structured files are self-contained logs of operations to individual files and are easily manipulated with tools outside of the context of an entire file system. For example, performing a diff on the history of two log-structured file streams is a cheap operation.

We do not believe that log-structured file streams are a drop-in replacement for whole-disk snapshotting. It is still valuable to be able to reinstate an entire byte-equivalent disk at some point in the past and directly boot from that version. Instead, log-structured file streams are useful by providing a log of file-level operations in between major whole-disk snapshots. Especially for forensics which often requires detailed logs of operations in order to understand exactly how a compromise occurred. They are also useful for architecting a backup system with efficient indexing and querying whole-file versions.

5.5.2 Inferring File Versions with Optimistic File Snapshotting

Creating versions out of file-level update stream logs is as simple as recording important locations in the log. We could of course let humans manually mark points in the log associated with important events. More than likely, the human marking the log chooses important points in time,

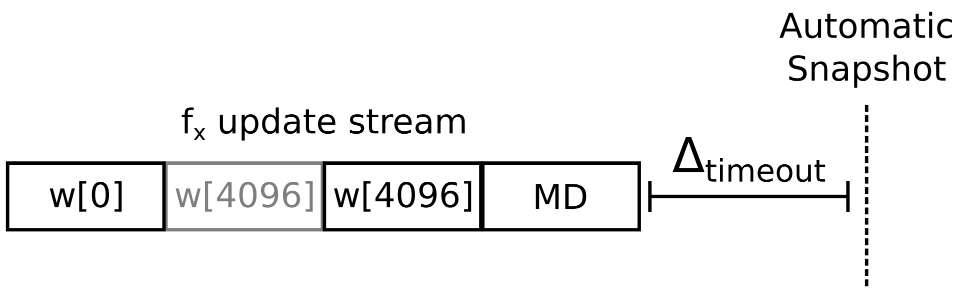


Figure 5.9: Files are optimistically snapshotted after waiting time Δ_{timeout}. MD stands for metadata, and the first write to position 4096 is gray because the second write supersedes it, representing an opportunity for garbage collection.

although it is possible that they inspect the file contents and make decisions based on the contents as well. However, ideally we would keep meaningful file versions without any human intervention whatsoever—completely automated. What type of semantics should we implement? Close-to-open semantics, as in AFS [49], map well into the expectations of users and are easy to understand. Every time you close a file a new version is created. However, with DS-VMI we do not have any insight into the *system calls* of a monitored system. We only have the writes, thus we must pick some heuristic for creating versions of files. We settled on a timeout since last write heuristic as an approximation to a close system call. Empirically it has been shown that cold files tend to remain cold over long time periods. Thus, as time passes the probability of an actual close increases.

Introspection provides the what—file-level updates—but not the how—system calls. This means that `/cloud-history` has no clear boundaries to perform versioning unlike a guest-supported versioning file system. In-guest agents have the luxury of leveraging system call information such as `open`'s and `close`'s to choose key versioning points for a given file. For example, many distributed file systems such as the Andrew File System (AFS) [49] have close-to-open semantics for consistency. `/cloud-history` has no such luxury. Ideally, versions of a file match at least each `open–close` pair. This follows what a normal user might expect: they `open` files to modify them, and `close` them signifying the "completion" of some task. Although valuable, understanding such `open–close` pairs is impossible with introspection.

Optimistic File Snapshotting determines when to version a file by waiting a tunable timeout, represented by Δ_{timeout} in Figure 5.9. This timeout is reset with every update to a file in the update stream provided by DS-VMI. As studies show [71], practically no files experience continuous updates. Thus, we can choose a timeout after which no more updates will occur with high probability. Snapshots are just positions in the log file, and work by replaying logs up till the requested version. Once created, snapshots are aggressively deduplicated and compressed at a byte level. Naturally, such an approach is prone to false positives—we may infer versions which never logically existed in the context of the user's environment. Without costly human annotation or intervention, it will be very difficult to ensure they are correct.

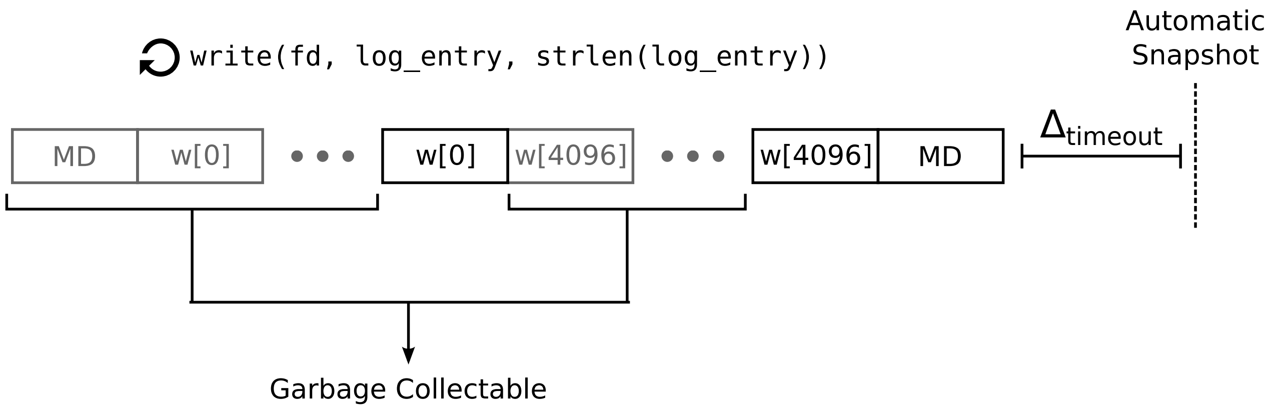


Figure 5.10: Garbage collection is especially important for append-only style workloads, such as those for log files. Here we see some guest process continuously writes lines into a log file. Blocks are written many times with redundant log entry data. Only the latest updates matter when a version is created.

5.5.3 Garbage Collecting Stale Block Writes

Garbage collection works well for workloads which repeatedly touch the same positions in files. For example, logging programs repeatedly write to the end of log files. These constant, append-only style workloads manifest themselves as many log entries recording written data to the same portion—the end—of a file over and over again. As a concrete example, imagine the last data block associated with a file receiving 10 updates, each representing the whole block, but with log file induced changes. A version of a log file need only record the last write to any given block. An illustration of such an append-only scenario is shown in Figure 5.10. Random-write workloads, such as those generated by a database, benefit less from garbage collection. Retention polices also determine the efficacy of garbage collection. A retention policy setting a very low timeout, or desiring a version on every update, reduces the benefit of garbage collection. A retention policy determining versions with a large timeout or over large time scales benefits more from garbage collection.

In `/cloud-history`, garbage collection is implemented as a scheduled task that runs asynchronously from the rest of the system. Garbage collection can be paused and restarted at a later point in time. Typically, garbage collection runs at the same time as generation of versions based on timeouts as mentioned in the last section, but it can technically run at any time. Between the versions of a file, garbage collection keeps the last write to the unique set of positions written to in that version, as well as the last metadata update to the unique set of metadata entries updated in that version. This new stream is written out as a compacted file-level update stream. The original stream, where new file-level updates arrive, is truncated.

The effects of garbage collection on three different styles of workloads are shown in Figure 5.10. Garbage collection is an optimization which helps scale the storage used for retaining history.

5.5.4 Whole-File Indexing and Deduplication

Because VMs running in a cloud are derived from a small set of OS and application configurations, we anticipate large amounts of file-level duplication. These results are confirmed with the NC State VM disk dataset, and also by our study of backup. One of the major goals of `/cloud-history` is to enable indexing over large amounts of historic state. As shown in Figure 5.3(a), even with just a base set of 140 images, we have over 4.5 million files. Indexers running over such a large corpus take significant amounts of time. However, as shown in Figure 5.4, running indexers over just the set of unique files takes an order of magnitude less time.

We expect this trend to be magnified with our versions of files through time. The reasoning is simple: all of these running VMs will receive the same updates, similar configuration changes if managed by a single tenant, and probably similar deployed software for any individual tenant. Often, individual tenants specialize on a certain software stack. For example, many web applications expect similar back ends such as the popular Linux, Apache, MySQL, PHP (LAMP) set of applications. Thus, many file-level update logs will contain the same versions of files repeated across many VMs. Our hunch is confirmed with a crawl over real backups from a small research cloud as shown in Figure 5.2. Unfortunately, because `/cloud-history` only exists as a prototype, we do not have long-term collections of logs like we do with the backup study, but what we learn from the 3,000+ snapshots is applicable to file-level update streams.

The attractiveness of a first-level index over unique files is that this form of index is OS- and application- agnostic—a design goal of `/cloud-history`. Yet, this OS- and application- agnostic index provides tremendous benefit to index-style workloads, precisely the type of workloads we want to support with `/cloud-history`.

Periodically, similar to garbage collection, the whole-file indexer executes and computes incremental hashes over all log-structured file streams. Incremental hashes are saved at each versioning point into a database for quick retrieval. Thus, each version is still kept on-disk and only a deduplicated index kept in the database. We assume that underlying storage technologies will capture this duplication via block-level deduplication. Thus, our whole-file deduplication index is maintained solely to cut down the computation time of indexing.

5.5.5 Block-Level Deduplication and Compression

We expect many duplicate files across VMs especially when tasks such as updates execute. We also expect duplicated files within VMs as some applications write to a temporary file and then atomically rename the file to its permanent location. Thus, supporting block-level deduplication to save storage space across multiple files and multiple disks is important. We could implement complex pointers directly in the log format. However, keeping these pointers up-to-date during events such as garbage collection and log compaction unnecessarily complicates the logic required to manage file-level update stream logs. Hence, we assume underlying storage technology performs block-level deduplication. As long as the file-level update stream log logic stores writes on block boundaries

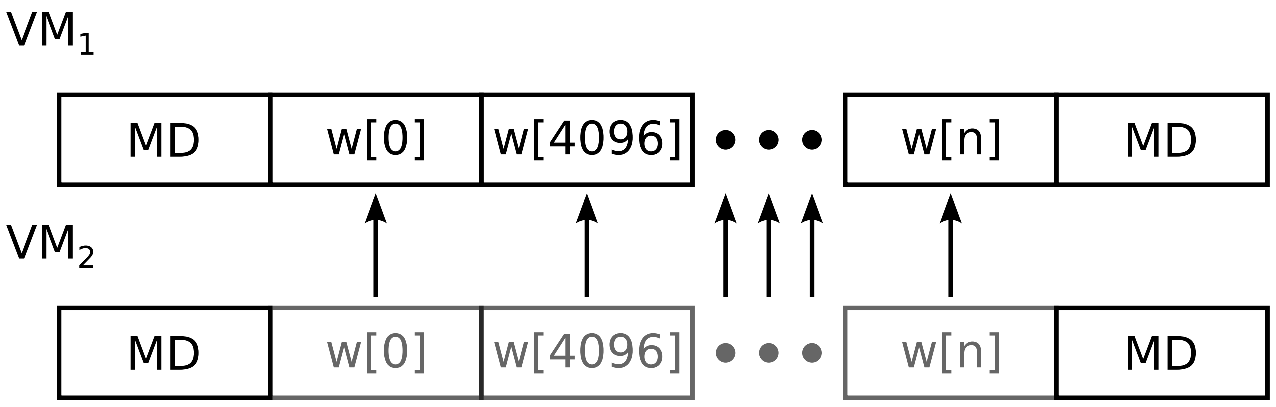


Figure 5.11: This figure shows two VMs (notionally, they could be the same VM) writing the same file. Their metadata which includes timestamps probably differs, but the data of the files is identical. Block-level deduplication is necessary to reclaim this wasted storage space. Also note that the metadata updates are often small and highly compressible.

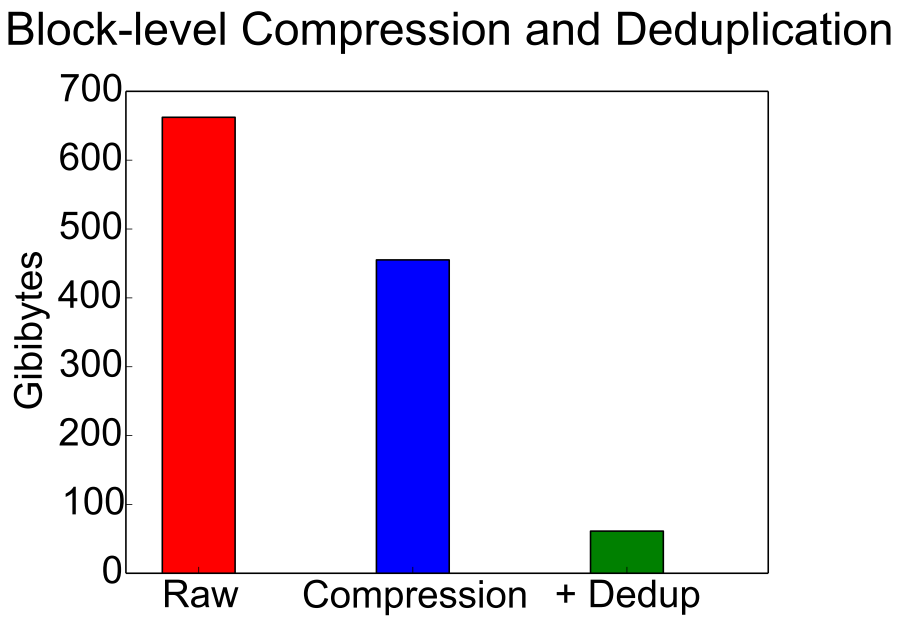


Figure 5.12: Shown are the effects of applying compression (LZ4) and block-level deduplication on the NC State dataset via ZFS.

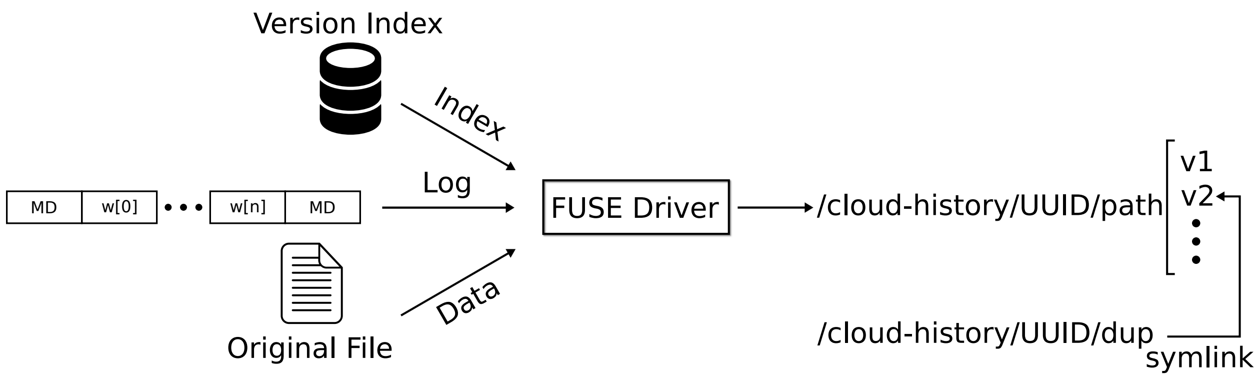


Figure 5.13: `/cloud-history` exposes versions of files via a simple synthetic FUSE file system which gives legacy indexers access to historic state without being rewritten. This figure shows that reads may come from the original file, or the file-level update stream log. Duplicate file versions are symlinked to a canonical version.

matching the blocks of the underlying storage platform, block-level deduplication proceeds without complication.

While it is valuable to have whole-file deduplication, research has shown that block-level deduplication is extremely valuable in the context of backups. Thus, we architected the log-structured on-disk layout be friendly to block-level deduplication. We assume that all of our file-level update streams are stored in storage that supports block-level deduplication. Modern file systems for example, ZFS or btrfs, both support block-level deduplication. We structured the log-file update stream's on-disk layout such that data blocks are aligned to the blocks of the underlying storage.

This means that given underlying storage that efficiently implements block-level deduplication and compression, `/cloud-history` automatically benefits from these savings. As an example of the benefits of block-level deduplication over whole-file deduplication see Figure 5.12.

5.6 Reconstructing File Versions

When an index does not exist for a specific query, the search mechanism needs efficient direct access to object-level contents. This means queries need to run as close as possible to the data to reduce as many bottlenecks as possible. In addition, many objects may be duplicated due to backups containing duplicates over time as well as across backed up systems running similar environments. Reducing the number of objects searched through is an important capability to make such search tractable [5, 96, 97, 105]. Modern backup systems can contain trillions of objects, but duplication often accounts for 90% or more of the storage space [110, 123].

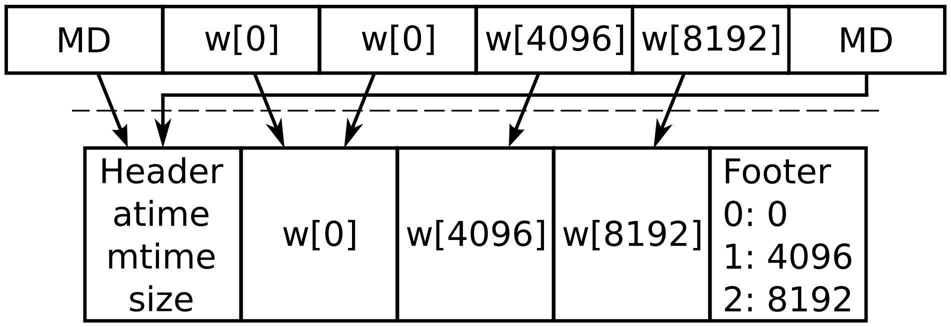


Figure 5.14: This shows the final version format after garbage collection, hashing, and timeout policies are applied. Note that metadata updates are coalesced in the header, and data updates are coalesced into a distinct set of writes to each block of a file. Each part of a version takes up a single on-disk block.

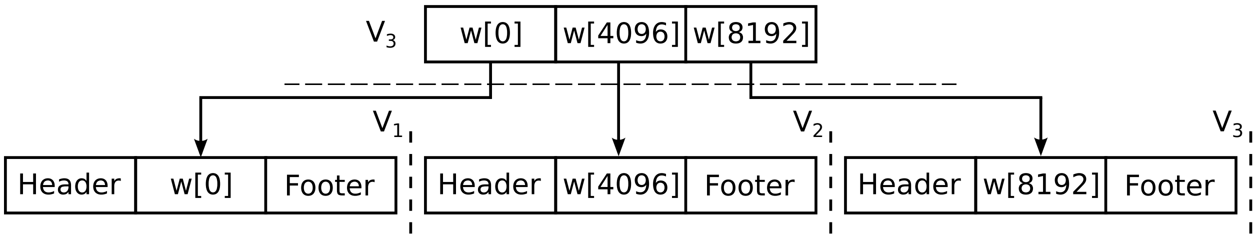


Figure 5.15: Reading files means reconstructing state from potentially multiple historic versions.

5.6.1 Efficient Object-Level Access

A backup system must minimize the overhead involved in accessing objects at the granularity of user queries. Overhead in accessing objects directly affects overall backup query time. The mismatch between the granularity of a query and the granularity of the backup system creates a fixed overhead. Ideally, this overhead is zero. In reality, the granularity of all future user queries is unpredictable. There always exists hypothetical queries with pathological overhead for any backup system. Thus, a tension arises between the level of granularity to store objects and the ease of implementing that granularity.

For example, on one extreme a backup system could implement sub-file, record-level indexing. This results in an explosion of metadata to track, as well as difficult maintenance. File formats experience significant churn due to tweaking or new emerging standards. On the other hand, a backup system could implement whole disk, block-level tracking. This results in minimal indexing and metadata tracking, and is extremely simple to maintain over time. Simplicity has led to this method being the default backup strategy in many backup systems.

We believe that whole-file granularity is the right granularity for a backup system to implement because it minimizes overhead to accessing objects, except at the record-level, and it is not difficult to maintain. Whole-file granularity requires indexing file systems. This does not pose a problem because file system on-disk data structures and layout change much more slowly than internal file formats. In addition, changes to file systems are often backwards-compatible. We therefore expect low long-term maintenance costs for file-system parsing modules. In addition, most modern file systems have excellent open source drivers and tools. Thus, whole-file granularity comes at low cost to the backup system.

5.6.2 Arbitrary Query Search

We cannot predict all potential future user queries. Thus, a backup system must maximize the freedom of query expression. Future potential queries range from the complex, "find all binaries vulnerable to a new zero-day vulnerability," to the simple, "find the latest version of my accidentally deleted document." Simple queries are generally handled well by most modern backup systems. This is because all backup systems have to track the timestamps of backups. They can therefore trivially serve simple queries looking for the latest version of important data.

Complex queries require carefully thought out architecture to enable high scalability and minimize query time. In-situ computation over backup data provides the highest form of scalability by discarding objects as early as possible [50]. Early discard minimizes wasted bandwidth and reduces overall backup system load. The query mechanism must also support arbitrary expressiveness by allowing any possible query from a user. Ideally, a user expresses queries using tools and environments familiar to them. For example, a DBA wants to express a historic query using SQL within a GUI-based client familiar to him or her. The ideal backup system fluidly conforms itself to user-familiar interfaces. Such a backup system would disrupt the status quo which forces conformance to the limited, often proprietary, interfaces that are generally provided. User-familiar interfaces require no training, and no time wasted or fidelity lost translating a query from a domain-specific to a domain-generic interface.

5.6.3 Evaluating Inferred File Versions

Figure 5.16 shows what happens with no guest coordination when versioning every 30 seconds. Only two versions match out of 12-13 versions. This implies a mismatch between the frequency of guest `sync` operations and the versioning of files. Fundamentally, we can only match versions across VMs if we match the underlying `sync`. This means that we must version at least twice as quickly as the highest frequency `sync` that we wish to match.

5.7 Securing Search

For the entirety of this chapter, we made the assumption that the user trusted their cloud with their raw backup data. What if the user does not want to trust the cloud? What if they want to store encrypted backups? We envisioned this case, and explored [97] architectures supporting

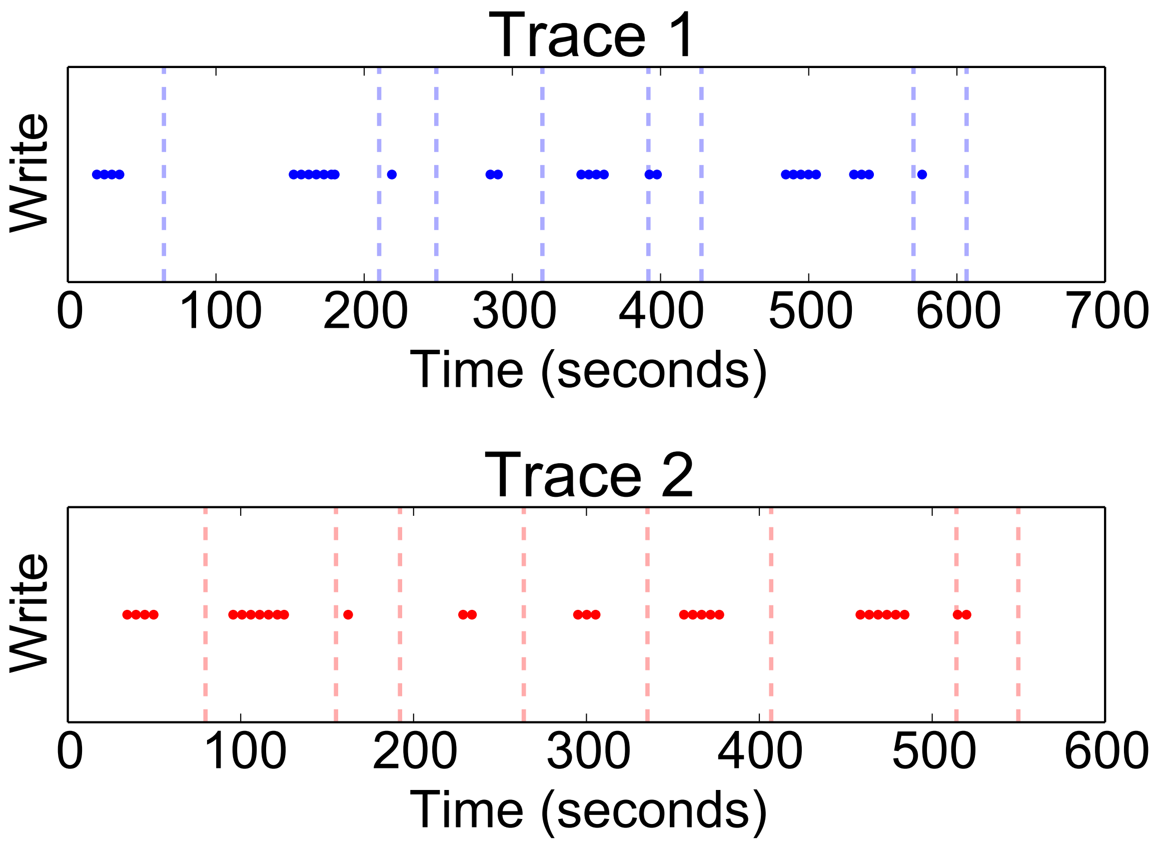


Figure 5.16: The arrival times of writes to a file without using `sync()` from within the guest OS. With an aggressively synchronizing guest OS, only 50% of the file versions matched. This demonstrates the worst case, that versions might not match always. Note that the history was compressible, for both traces we collected between 42 - 47 MiB which compressed down into 900 - 700 KiB.

it via convergent encryption [35]. At the time of indexing, users provide keys to decrypt stored objects—presumably files, although they can be at any granularity. We decided on using a convergent encryption scheme per-user as that allows for file-level deduplication without revealing the contents of the files. Each cloud user can key their convergent encryption algorithms with different keys, thus it would be impossible for a cloud to know if one user has the same files as another user. Frequency analysis would still be possible, and this means the only truly secure method requires abandoning any deduplication index. If users deem that tradeoff worth it, they are free to use non-convergent cryptography with the understanding that indexing becomes very costly, potentially intractable depending on the number of and sizes of files.

We explored three distinct architectures for indexing user-encrypted data: (1) one in which users trust the cloud, (2) one in which users trust the cloud and a key escrow service, and (3) one in which users trust no third party with encryption keys. If users trust the cloud, then as the cloud versions file-level update logs it encrypts the blocks differentiating each version with a key derived via a hash of those blocks. Thus, if a file experienced 10 block updates in a new version, those 10 blocks would be convergently encrypted with their hash and that hash stored encrypted using a key provided by the user. The cloud saves file version keys, but users control their key used to encrypt them. If users do not trust the cloud, but they do trust a key escrow service, they can convergently encrypt their files before they are saved to disk. This still exposes duplicate files to the cloud, but leaves the keys out of its reach. The key escrow service keeps keys for users, and performs decryption operations when the cloud indexes their files. In the last setting, the user trusts no third parties. In this case they still convergently encrypt their files before saving them to disk, but they also keep the keys instead of sharing them. If they want their files further indexed by the cloud, they must present the per-file keys at indexing time. Alternatively, users could index their files using their own compute instances pulling encrypted files from `/cloud-history`.

As a simple initial implementation, we allow the user to provide decryption keys when beginning their indexing or search operation. Thus, at search or index time, the user sends a list of 3-tuples: $< pathname,\ encryption\ method,\ key >$. This list is used to retrieve files from backup storage before they are indexed or searched. This mechanism could be extended to be less verbose and cumbersome for searches of large scope. This requires striking a balance between usability, privacy and performance. At one extreme is a single encryption key for an entire VM image. The other extreme (our initial choice) is a key per file within a VM image. A hierarchical file system within a VM image offers natural directory-level or subtree-level aggregation possibilities for intermediate points of this spectrum. This would require augmenting the 3-tuples mentioned above with an element denoting the granularity of the decryption key: $< granularity,\ pathname,\ encryption\ method,\ key >$, with possible granularity values of "VM image," "subtree," or "file." In addition, we are exploring the option of giving users direct control of the search infrastructure in the cloud for their searches such that they never reveal keys to any other party. This final path maintains privacy while providing a service that scales with the scope of searches.