VSS: A Storage System for Video Analytics

Brandon Haynes
brandon.haynes@microsoft.com
Gray Systems Lab, Microsoft

Maureen Daum
mdaum@cs.washington.edu
University of Washington

Dong He
donghe@cs.washington.edu
University of Washington

Amrita Mazumdar
amrita@cs.washington.edu
University of Washington

Magdalena Balazinska
magda@cs.washington.edu
University of Washington

Alvin Cheung
akcheung@cs.berkeley.edu
University of California, Berkeley

Luis Ceze
luisceze@cs.washington.edu
University of Washington

ABSTRACT

We present a new video storage system (VSS) designed to decouple high-level video operations from the low-level details required to store and efficiently retrieve video data. VSS is designed to be the storage subsystem of a video data management system (VDBMS) and is responsible for: (1) transparently and automatically arranging the data on disk in an efficient, granular format; (2) caching frequently-retrieved regions in the most useful formats; and (3) eliminating redundancies found in videos captured from multiple cameras with overlapping fields of view. Our results suggest that VSS can improve VDBMS read performance by up to 54%, reduce storage costs by up to 45%, and enable developers to focus on application logic rather than video storage and retrieval.

ACM Reference Format:
Brandon Haynes, Maureen Daum, Dong He, Amrita Mazumdar, Magdalena Balazinska, Alvin Cheung, and Luis Ceze. 2021. VSS: A Storage System for Video Analytics. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3448016.3459242

1 INTRODUCTION

The volume of video data captured and processed is rapidly increasing: YouTube receives more than 400 hours of uploaded video per minute [52], and more than six million closed-circuit television cameras populate the United Kingdom, collectively amassing an estimated 7.5 petabytes of video per day [9]. More than 200K body-worn cameras are in service [24], collectively generating almost a terabyte of video per day [55].

To support this video data deluge, many systems and applications have emerged to ingest, transform, and reason about such data [19, 23, 25, 27, 28, 34, 43, 56]. Critically, however, most of these systems lack efficient storage managers. They focus on query execution for a video that is already decoded and loaded in memory [23, 27, 28] or treat video compression as a black box [25, 34, 56] (cf. [19, 43]). In practice, of course, videos are stored on disk, and the cost of reading and decompressing is high relative to subsequent processing [11, 19], e.g., constituting more than 50% of total runtime [29]. The result is a performance plateau limited by Amdahl's law, where an emphasis on post-decompression performance might yield impressive results in isolation, but ignores the diminishing returns when performance is evaluated end-to-end.

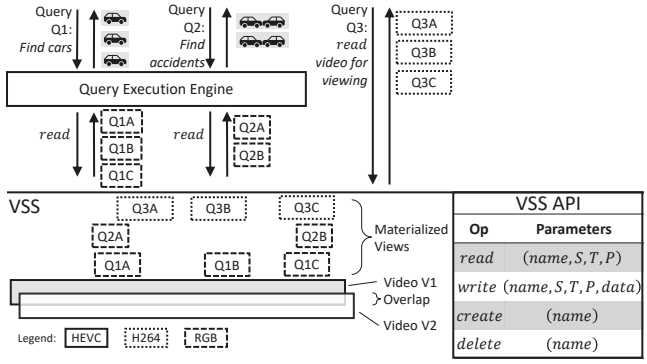


Figure 1: VSS overview & API. Reads and writes require specification of spatial (S; resolution, region of interest), temporal (T; start/end time, frame rate), and physical (P; frame layout, compression codec, quality) parameters.

In this paper, we develop VSS, a *video storage system* designed to serve as storage manager beneath a video data management system or video processing application (collectively VDBMSs). Analogous to a storage and buffer manager for relational data, VSS assumes responsibility for storing, retrieving, and caching video data. It frees higher-level components to focus on application logic, while VSS optimizes the low-level performance of video data storage. As we will show, this decoupling dramatically speeds up video processing queries and decreases storage costs. VSS does this by addressing the following three challenges:

First, modern video applications commonly issue multiple queries over the same (potentially overlapping) video regions and build on each other in different ways (e.g., Figure 1). Queries can also vary video resolution and other characteristics (e.g., the SMOL system

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3459242

rescales video to various resolutions [29] and Chameleon dynamically adjusts input resolution [25]). Such queries can be dramatically faster with an efficient storage manager that maintains and evolves a cache of video data, each differently compressed and encoded.

Second, if the same video is queried using multiple systems such as via a VDBMS optimized for simple select and aggregate queries [27] and a separate vision system optimized for reasoning about complex scenes [48] (e.g., Figure 1), then the video file may be requested at different resolutions and frame rates and using different encodings. Having a single storage system that encapsulates all such details and provides a unified query interface makes it seamless to create—and optimize—such federated workflows. While some systems have attempted to mitigate this by making multiple representations available to developers [49, 54], they expensively do so for entire videos even if only small subsets (e.g., the few seconds before and after an accident) are needed in an alternate representation.

Third, many recent applications analyze large amounts of video data with overlapping fields of view and proximate locations. For example, traffic monitoring networks often have multiple cameras oriented toward the same intersection and autonomous driving and drone applications come with multiple overlapping sensors that capture nearby video. Reducing the redundancies that occur among these sets of physically proximate or otherwise similar video streams is neglected in all modern VDBMSs. This is because of the substantial difficulties involved: systems (or users) need to consider the locations, orientations, and fields of view of each camera to identify redundant video regions; measure overlap, jitter, and temporally align each video; and ensure that deduplicated video data can be recovered with sufficient quality. Despite these challenges, and as we show herein, deduplicating overlapping video data streams offers opportunities to greatly reduce storage costs.

VSS addresses the above challenges. As a storage manager, it exposes a simple interface where VDBMSs read and write videos using VSS's API (see Figure 1). Using this API, systems write video data in any format, encoding, and resolution—either compressed or uncompressed—and VSS manages the underlying compression, serialization, and physical layout on disk. When these systems subsequently read video—once again in any configuration and by optionally specifying regions of interest and other selection criteria—VSS automatically identifies and leverages the most efficient methods to retrieve and return the requested data.

VSS deploys the following optimizations and caching mechanisms to improve read and write performance. First, rather than storing video data on disk as opaque, monolithic files, VSS decomposes video into sequences of contiguous, independently-decodable sets of frames. In contrast with previous systems that treat video as static and immutable data, VSS applies transformations at the granularity of these sets of frames, freely transforming them as needed to satisfy a read operation. For example, if a query requests a video region compressed using a different codec, VSS might elect to cache the transcoded subregion and delete the original.

As VSS handles requests for video over time, it maintains a per-video on-disk collection of materialized views that is populated passively as a byproduct of read operations. When a VDBMS performs a subsequent read, VSS leverages a minimal-cost subset of these views to generate its answer. Because these materialized views can arbitrarily overlap and have complex interdependencies, finding the least-cost set of views is non-trivial. VSS uses a satisfiability modulo theories (SMT) solver to identify the best views to satisfy a request. VSS prunes stale views by selecting those least likely to be useful in answering subsequent queries. Among equivalently useful views, VSS optimizes for video quality and defragmentation.

Finally, VSS reduces the storage cost of redundant video data collected from physically proximate cameras. It does so by deploying a *joint compression* optimization that identifies overlapping regions of video and stores these regions only once. The key challenge lies in efficiently identifying potential candidates for joint compression in a large database of videos. Our approach identifies candidates efficiently without requiring any metadata specification. To identify video overlap, VSS incrementally fingerprints video fragments (i.e., it produces a feature vector that robustly characterizes video regions) and, using the resulting fingerprint index, searches for likely correspondences between pairs of videos. It finally performs a more thorough comparison between likely pairs.

In summary, we make the following contributions:

- We design a new storage manager for video data that leverages the fine-grained physical properties of videos to improve application performance (Section 2).
- We develop a novel technique to perform reads by selecting from many materialized views to efficiently produce an output while maintaining the quality of the resulting video data (Section 3).
- We develop a method to optimize the storage required to persist videos that are highly overlapping or contain similar visual information, an indexing strategy to identify such regions (Section 5), and a protocol for caching multiple versions of the same video (Section 4).

We evaluate VSS against existing video storage techniques and show that it can reduce video read time by up to 54% and decrease storage requirements by up to 45% (Section 6).

2 VSS OVERVIEW

Consider an application that monitors an intersection for automobiles associated with missing children or adults with dementia. A typical implementation would first ingest video data from multiple locations around the intersection. It would then index regions of interest, typically by decompressing and converting the entire video to an alternate representation suitable for input to a machine learning model trained to detect automobiles. Many video query processing systems provide optimizations that accelerate this process [27, 35, 54]. Subsequent operations, however, might execute more specific queries only on the regions that have automobiles. For example, if a red vehicle is missing, a user might issue a query to identify all red vehicles in the dataset. Afterward, a user might request and view all video sequences containing only the likely candidates. This might involve further converting relevant regions to a representation compatible with the viewer (e.g., at a resolution compatible with a mobile device or compressed using a supported codec). We show VSS's performance for this application in Section 6.

While today's video processing engines perform optimizations for operations over entire videos (e.g., the indexing phase described above), their storage layers provide little or no support for subsequent queries over the results (even dedicated systems such as quFiles [49] or VStore [54] transcode entire videos, even when only a few frames are needed). Meanwhile, when the above application uses VSS to read a few seconds of low-resolution, uncompressed video data to find frames containing automobiles, it can delegate responsibility to VSS for efficiently producing the desired frames. This is true even if the video is streaming or has not fully been written to disk.

Critically, VSS automatically selects the most efficient way to generate the desired video data in the requested format and region of interest (ROI) based on the original video and cached representations. Further, to support real-time streaming scenarios, writes to VSS are non-blocking and users may query prefixes of ingested video data without waiting on the entire video to be persisted.

Figure 1 summarizes the set of VSS-supported operations. These operations are over *logical videos*, which VSS executes to produce or store fine-grained *physical video* data. Each operation involves a point- or range-based scan or insertion over a single logical video source. VSS allows constraints on combinations of temporal (T), spatial (S), and physical (P) parameters. Temporal parameters include start and end time interval ($[s, e]$) and frame rate (f); spatial parameters include resolution ($r_x \times r_y$) and region of interest ($[x_0..x_1]$ and $[y_0..y_1]$); and physical parameters P include physical frame layout (l; e.g., YUV420, YUV422), compression method (c; e.g., HEVC), and quality (to be discussed in Section 3.2).

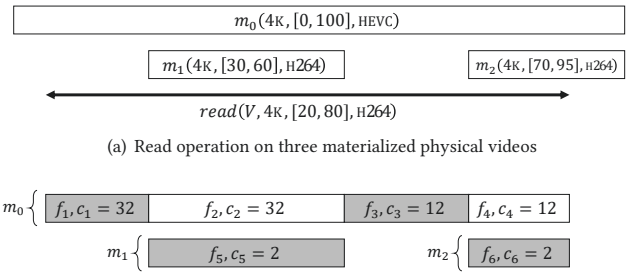
Internally, VSS arranges each written physical video as a sequence of entities called *groups of pictures (GOPs)*. Each GOP is composed of a contiguous sequence of frames in the same format and resolution. A GOP may include the full frame extent or be cropped to some ROI and may contain raw pixel data or be compressed. Compressed GOPs, however, are constrained such that they are independently decodable and take no data dependencies on other GOPs.

Though a GOP may contain an unbounded number of frames, video compression codecs typically fix their size to a small, constant number of frames (30–300) and VSS accepts as-is ingested compressed GOP sizes (which are typically less than 512kB). For uncompressed GOPs, our prototype implementation automatically partitions video data into blocks of size $\leq$ 25MB (the size of one RGB 4K frame), or a single frame for resolutions that exceed this threshold.

3 DATA RETRIEVAL FROM VSS

As mentioned, VSS internally represents a logical video as a collection of materialized physical videos. When executing a read, VSS produces the result using one or more of these views.

Consider a simplified version of the application described in Section 2, where a single camera has captured 100 minutes of 4K resolution, HEVC-encoded video, and written it to VSS using the name V. The application first reads the entire video and applies a computer vision algorithm that identifies two regions (at minutes 30–60 and 70–95) containing automobiles. The application then retrieves those fragments compressed using H264 to transmit to a device that only supports this format. As a result of these operations, VSS now contains the original video (m_0) and the cached versions of



(a) Read operation on three materialized physical videos

(b) Physical video fragments with simplified cost formulae

Figure 2: Figure 2(a) shows the query $read(V, 4K, [20, 80], H264)$, where VSS has materialized m_0, m_1, and m_2. Figure 2(b) shows weighted fragments and costs. The lowest-cost result is shaded.

the two fragments (m_1, m_2) as illustrated in Figure 2(a). The figure indicates the labels $\{m_0, m_1, m_2\}$ of the three videos, their spatial configuration (4K), start and end times (e.g., [0, 100] for m_0), and physical characteristics (HEVC or H264).

Later, a first responder on the scene views a one-hour portion of the recorded video on her phone, which only has hardware support for H264 decompression. To deliver this video, the application executes $read(V, 4K, [20, 80], H264)$, which, as illustrated by the arrow in Figure 2(a), requests video V at 4K between time [20, 80] compressed with H264.

VSS responds by first identifying subsets of the available physical videos that can be leveraged to produce the result. For example, VSS can simply transcode m_0 between times [20, 80]. Alternatively, it can transcode m_0 between time [20, 30] and [60, 70], m_1 between [30, 60], and m_2 between [70, 80]. The latter plan is the most efficient since m_1 and m_2 are already in the desired output format (H264), hence VSS need not incur high transcoding costs for these regions. Figure 2(b) shows the different selections that VSS might make to answer this read. Each *physical video fragment* $\{f_1, ..f_6\}$ in Figure 2(b) represents a different region that VSS might select. Note that VSS need not consider other subdivisions—for example by subdividing f_5 at [30, 40] and [40, 60]—since f_5 being cheaper at [30, 40] implies that it is at [40, 60] too.

To model these transcoding costs, VSS employs a *transcode cost model* $c_t(f, P, S)$ that represents the cost of converting a physical video fragment f into a target spatial and physical format S and P. The selected fragments must be of sufficient quality, which we model using a *quality model* $u(f, f')$ and reject fragments of insufficient quality. We introduce these models in the following two subsections.

3.1 Cost Model

We first discuss how VSS selects fragments for use in performing a read operation using its cost model. In general, given a *read* operation and a set of physical videos, VSS must first select fragments that cover the desired spatial and temporal ranges. To ensure that a solution exists, VSS maintains a *cover* of the initially-written video m_0 consisting of physical video fragments with quality equal to the original video (i.e., $u(m_0, f) \geq \tau$). Our prototype

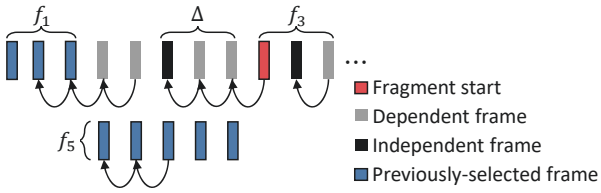


Figure 3: A simplified illustration based on Figure 2. VSS has decided to use f_1 and f_5 and is considering using f_3 starting at the red-highlighted frame. However, f_3 cannot be decoded without transitively decoding its dependencies shown by directed edges (labeled Δ).

sets a threshold $\tau = 40$dB, which is considered to be lossless. See Section 3.2 for details. VSS also returns an error for reads extending outside of the temporal interval of m_0.

Second, when the selected physical videos temporally overlap, VSS must resolve *which* physical video fragments to use in producing the answer in a way that minimizes the total conversion cost of the selected set of video fragments. This problem is similar to materialized view selection [15]. Fortunately, a VSS read is far simpler than a general database query, and in particular is constrained to a small number of parameters with point- or range-based predicates.

We motivate our solution by continuing our example from Figure 2(a). First, observe that the collective start and end points of the physical videos form a set of *transition points* where VSS can switch to an alternate physical video. In Figure 2(a), the transition times include those in the set $\{30, 60, 70\}$, and we illustrate them in Figure 2(b) by partitioning the set of cached materialized views at each transition point. VSS ignores fragments that are outside the read's temporal range, since they do not provide information relevant to the read operation.

Between each consecutive pair of transition points, VSS must choose exactly one physical video fragment. In Figure 2(b), we highlight one such set of choices that covers the read interval. Each choice of a fragment comes with a cost (e.g., f_1 has cost 32), derived using a cost formula given by $c_t(f, P, S) = \alpha(f_S, f_P, S, P) \cdot |f|$. This cost is proportional to the total number of pixels $|f|$ in fragment f scaled by $\alpha(S, P, S', P')$, which is the normalized cost of transcoding a single pixel from spatial and physical format (S, P) into format (S', P'). For example, using fragment m_1 in Figure 2 requires transcoding from physical format $P = \text{HEVC}$ to $P' = \text{H264}$ with no change in spatiotemporal format (i.e., $S = S'$).

During installation, VSS computes the domain of α by executing the vbench benchmark [31] on the installation hardware, which produces per-pixel transcode costs for a variety of resolutions and codecs. For resolutions not evaluated by vbench, VSS approximates α by piecewise linear interpolation of the benchmarked resolutions.

VSS must also consider the data dependencies between frames. Consider the illustration in Figure 3, which shows the frames within a physical video with their data dependencies indicated by directed edges. If VSS wishes to use a fragment at the frame highlighted in red, it must first decode all of the red frame's *dependent frames*, denoted by the set Δ in Figure 3. This implies that the cost of transcoding a frame depends on *where* within the video it occurs, and whether its dependent frames are *also* transcoded.

To model this, we introduce a *look-back cost* $c_l(\Omega, f)$ that gives the cost of decoding the set of frames Δ on which fragment f depends *if they have not already been decoded*, meaning that they are not in the set of previously selected frames Ω. As illustrated in Figure 3, these dependencies come in two forms: independent frames $A \subseteq \Delta$ (i.e., frames with out-degree zero in our graphical representation) which are larger in size but less expensive to decode, and the remaining dependent frames $\Delta - A$ (those with outgoing edges) which are highly compressed but have more expensive decoding dependencies between frames. We approximate these per-frame costs using estimates from Costa et al. [10], which empirically concludes that dependent frames are approximately 45% more expensive than their independent counterparts. We therefore fix $\eta = 1.45$ and formalize look-back cost as $c_l(\Omega, f) = |A - \Omega| + \eta \cdot |(\Delta - A) - \Omega|$.

To conclude our example, observe that our goal is to choose a set of physical video fragments that cover the queried spatiotemporal range, do not temporally overlap, and minimize the decode and look-back cost of selected fragments. In Figure 2(b), of all the possible paths, the one highlighted in gray minimizes this cost. These characteristics collectively meet the requirements identified at the beginning of this section.

Generating a minimum-cost solution using this formulation requires jointly optimizing both look-back cost c_l and transcode cost c_t, where each fragment choice affects the dependencies (and hence costs) of future choices. These dependencies make the problem not solvable in polynomial time, and VSS employs an SMT solver [12] to generate an optimal solution. Our embedding constrains frames in overlapping fragments so that only one is chosen, selects combinations of regions of interest (ROI) that spatially combine to cover the queried ROI, and uses information about the locations of independent and dependent frames in each physical video to compute the cumulative decoding cost due to both transcode and look-back for any set of selected fragments. We compare this algorithm to a dependency-naïve greedy baseline in Section 6.1.

3.2 Quality Model

Besides efficiency, VSS must also ensure that the quality of a result has sufficient fidelity. For example, using a heavily *downsampled* (e.g., 32×32 pixels) or *compressed* (e.g., at a 1Kbps bitrate) physical video to answer a read requesting 4K video is likely to be unsatisfactory. VSS tracks quality loss from both sources using a quality model $u(f_0, f)$ that gives the expected quality loss of using a fragment f in a read operation relative to using the originally-written video f_0. When considering using a fragment f in a read, VSS will reject it if the expected quality loss is below a user-specified cutoff: $u(f_0, f) < \epsilon$. The user optionally specifies this cutoff in the read's physical parameters (see Figure 1); otherwise, a default threshold is used ($\epsilon = 40$dB in our prototype).

The range of u is a non-negative peak signal-to-noise ratio (PSNR), a common measure of quality variation based on mean-squared error [22]. Values ≥40dB are considered to be lossless qualities, and ≥30dB near-lossless. PSNR is itself defined in terms of the mean-squared error (MSE) of the pixels in a frame relative to the corresponding pixels in a reference frame, normalized by the maximum pixel value.

As described previously, error in a fragment accumulates through two mechanisms—resampling and compression—and VSS uses the sum of both sources when computing u. We next examine how VSS computes error from each source.

Resampling error. First, for downsampled error produced through a resolution or frame rate change applied to f_0, computing $MSE(f, f_0)$ is straightforward. However, VSS may transitively apply these transformations to a sequence of fragments. For example, f_0 might be downsampled to create f_1, and f_1 later used to produce f_2. In this case, when computing $MSE(f_0, f_2)$, VSS no longer has access to the uncompressed representation of f_0. Rather than expensively re-decompressing f_0, VSS instead bounds $MSE(f_0, f_n)$ in terms of $MSE(f_0, f_1)$, ..., $MSE(f_{n-1}, f_n)$, which are a single real-valued aggregates stored as metadata. We show a proof in [17].

Compression error. Unlike resampling error, tracking quality loss due to lossy compression error is challenging because it cannot be calculated without decompressing—an expensive operation—and comparing the recovered version to the original input. Instead, VSS estimates compression error in terms of mean bits per pixel per second (MBPP/S), which is a metric reported during (re)compression. VSS then estimates quality by mapping MBPP/S to the PSNR reported by the vbench benchmark [31], a benchmark for evaluating video transcode performance in the cloud. To improve on this estimate, VSS periodically samples regions of compressed video, computes exact PSNR, and updates its estimate.

4 DATA CACHING IN VSS

We now describe how VSS decides *which* physical videos to maintain, and which to evict under low disk space conditions. This involves making two interrelated decisions:

- When executing a read, should VSS admit the result as a new physical video for use in answering future reads?
- When disk space grows scarce, which existing physical video(s) should VSS discard?

To aid both decisions, VSS maintains a video-specific *storage budget* that limits the total size of the physical videos associated with each logical video. The storage budget is set when a video is created in VSS (see Figure 1) and may be specified as a multiple of the size of the initially written physical video or a fixed ceiling in bytes. This value is initially set to an administrator-specified default (10× the size of the initially-written physical video in our prototype). As described below, VSS ensures a sufficiently-high quality version of the original video can always be reproduced. It does so by maintaining a cover of fragments with sufficiently high quality (PSNR ≥ 40dB in our prototype, which is considered to be lossless) relative to the originally ingested video.

The key idea behind VSS's cache is to logically break physical videos into "pages." That is, rather than treating each physical video as a monolithic cache entry, VSS targets the individual GOPs *within* each physical video. Using GOPs as cache pages greatly homogenizes the sizes of the entries that VSS must consider. VSS's ability to evict GOP pages *within* a physical video differs from other variable-sized caching efforts such as those used by content delivery networks (CDNs), which make decisions on large, indivisible, and opaque entries (a far more challenging problem space with limited solutions [7]).

However, there are key differences between GOPs and pages. In particular, GOPs are related to each other; i.e., (i) one GOP might be a higher-quality version of another, and (ii) consecutive GOPs form a contiguous video fragment. These correlations make typical eviction policies like least-recently used (LRU) inefficient. In particular, naïve LRU might evict every other GOP in a physical video, decomposing it into many small fragments and increasing the cost of reads (which have exponential complexity in the number of fragments).

Additionally, given multiple, redundant GOPs that are all variations of one another, ordinary LRU would treat eviction of a redundant GOP the same as any other GOP. However, our intuition is that it is desirable to treat redundant GOPs different than singleton GOPs without such redundancy.

Given this intuition, VSS employs a modified LRU policy (LRU_{VSS}) that associates each fragment with a nonnegative sequence number computed using ordinary LRU offset by:

- **Position (p).** To reduce fragmentation, VSS increases the sequence number of fragments near the middle of a physical video, relative to the beginning or end. For a video with n fragments arranged in ascending temporal order, VSS increases the sequence number of fragment f_i by $p(f_i) = \min(i, n - i)$.
- **Redundancy (r).** VSS decreases the sequence number of fragments that have redundant or higher-quality variants. To do so, using the quality cost model u, VSS generates a u-ordering of each fragment f_i and all other fragments that are a spatiotemporal cover of f_i. VSS decreases the sequence number of f_i by its rank $r(f_i) : \mathbb{Z}^{0+}$ in this order (i.e., $r(f_i) = 0$ for a fragment with no higher-quality alternatives, while $r(f_i) = n$ for a fragment with n higher-quality variants).
- **Baseline quality (b).** VSS never evicts a fragment if it is the only fragment with quality equal to the quality of the corresponding fragment m_0 in the originally-written physical video. To ensure this, given a set of fragments F in a video, VSS increases the sequence number of each fragment by (our prototype sets $\tau = 40$):

$$b(f_i) = \begin{cases} +\infty & \text{if } \nexists f_j \in F \setminus f_i \,.\, u(m_0, f_j) \geq \tau \\ 0 & \text{otherwise} \end{cases}$$

Using the offsets described above, VSS computes the sequence number of each candidate fragment f_i as $LRU_{\text{vss}}(f_i) = LRU(f_i) + \gamma \cdot p(f_i) - \zeta \cdot r(f_i) + b(f_i)$. Here weights γ and ζ balance between position and redundancy, and our prototype weights the former ($\gamma = 2$) more heavily than the latter ($\zeta = 1$). It would be a straightforward extension to expose these as parameters tunable for specific workloads.

5 DATA COMPRESSION IN VSS

VSS employs two compression-oriented optimizations and one optimization that reduces the number of physical video fragments. Specifically, VSS (i) jointly compresses redundant data across multiple physical videos (Section 5.1); (ii) lazily compresses blocks of uncompressed, infrequently-accessed GOPs (Section 5.2); and (iii) improves the read performance by compacting temporally-adjacent video (Section 5.3).