VSS: A Database Storage System for Video Analytics

ABSTRACT

We present a new video storage system (VSS) designed to decouple high-level video operations such as machine learning and computer vision from the low-level details required to store and efficiently retrieve video data. Using VSS, video data management systems (VDBMSs) read and write video data from VSS, letting it be 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.

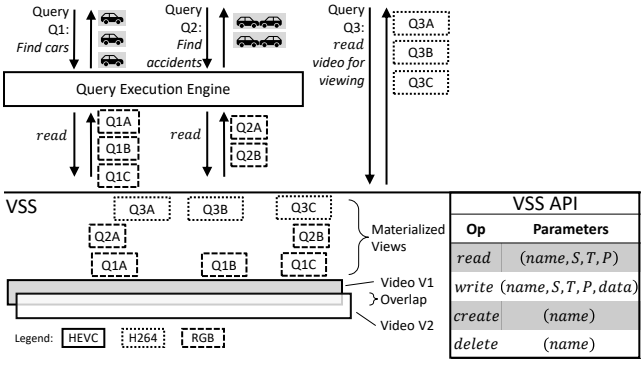


Figure 1: VSS overview & API. Reads and writes require specification of spatial (S; resolution, region of interest), temporal (T; start/end time, frame rate), and physical (P; frame layout, compression codec, quality) parameters.

1 INTRODUCTION

Driven by advances in computer vision and machine learning, along with the proliferation of cameras in the world around us, the volume of video data captured and processed is rapidly increasing. YouTube receives more than 400 hours of uploaded video per minute [55], and more than six million closed-circuit television cameras populate the United Kingdom, collectively amassing an estimated 7.5 petabytes of video data per day [9]. More than 200,000 law-enforcement body-worn cameras are in service [25], collectively generating almost a terabyte of video per day [59].

To support this video data deluge, many systems and applications have emerged to ingest, transform, and reason about such data [19, 23, 27, 29, 30, 36, 45, 60]. Critically, however, most of these systems lack efficient storage managers. They focus on query execution given a video that is already decoded and loaded in memory [23, 29, 30] or treat video compression as an opaque black box [27, 36, 60] (cf. [19, 45]). In practice, of course, videos are stored on disk, and the cost of reading videos from disk and decoding them are known to be high relative to subsequent processing [11, 19], e.g., constituting more than 50% of total runtime [31]. 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 completely ignores the rapidly diminishing returns obtained when performance is evaluated end-to-end.

In this paper, we develop VSS, a *video storage system* designed as an efficient storage system for video analytics applications. VSS is 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 systems to focus on application logic, while VSS optimizes the low-level performance of video data. As we will show, this decoupling dramatically speeds up video processing queries and decreases storage costs. VSS addresses 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., SMOL rescales video to various resolutions [31] and Chameleon dynamically adjusts input resolution [27]). Such queries can be dramatically faster with an efficient storage manager that maintains and evolves a cache of materialized views, each differently compressed and encoded.

Second, if the same video is queried using multiple systems such as a VDBMS optimized for simple select and aggregate queries [29] and a separate vision system optimized for reasoning about complex scenes [51] (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 [52, 58] by making multiple representations available to developers, 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: users (or systems) 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 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 potentially 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 and 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 ingest video data from multiple locations around the intersection, decompress it, and convert it to an alternate representation suitable for input to a machine learning model trained to detect automobiles or license plates. Such models typically use deep learning and inference is expensive. Therefore, the application might first use an inexpensive low-resolution video representation to prune frames that are unlikely to contain an automobile at all. Subsequent operations might execute more specific queries on regions that do contain automobiles (e.g., to select automobiles of a particular color). Many video query processing systems provide such an optimization to accelerate queries [29, 37, 58]. Afterward, a user might request and view all video sequences containing 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 performance results for these operations under VSS in Section 6.

While today's video processing engines perform the above optimizations at the query execution layer, their storage layers provide little or no support for them (even dedicated systems such as quFiles [52] or VStore [58] transcode entire videos, even when only a few frames are needed at a particular resolution or configuration). On the other hand, when the above application uses VSS to read a few seconds of low-resolution, uncompressed video data to find frames with license plates, 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. For example, only certain regions of frames generally contain license plates. If the low-resolution heuristic identifies such a region, the application can explicitly request just that ROI. 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 full set of operations that VSS exposes. Importantly, 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 ≤ 25MB (the size of one RGB 4K frame), or a single frame for resolutions that exceed this threshold.

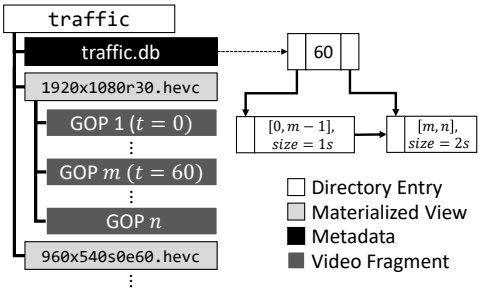
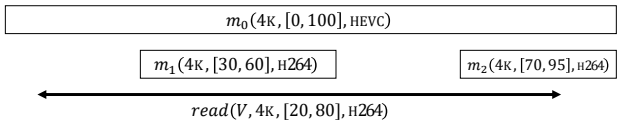
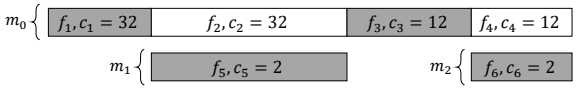


Figure 2: An example VSS physical organization that contains one logical video and two underlying physical videos. For physical video `1920x1080r30.hevc`, the first m GOPs are each one second in length, while the remaining $n - m$ are two seconds. These durations are recorded in the associated index.



(a) Read operation on three materialized physical videos



(b) Physical video fragments with simplified cost formulae

Figure 3: Figure 3(a) shows the query $read(V, 4K, [20, 80], H264)$ executed over a video with materialized videos $\{m_0, m_1, m_2\}$. Figure 3(b) shows the weighted physical video fragments using simplified cost formulae. The lowest-cost result is shaded.

Figure 2 illustrates the internal physical state of VSS. In this example, VSS contains a single logical video `traffic` with two physical representations (one HEVC at 1920 × 1080 resolution and 30 frames per second, and a 60-second variant at 960 × 540 resolution). VSS has stored the GOPs associated with each representation as a series of separate files (e.g., `traffic/1920x1080r30.hevc/1`). It has also constructed a non-clustered temporal index that maps time to the file containing associated visual information. This level of detail is invisible to applications, which access VSS only through the operations summarized in Figure 1.

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 license plate 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 license plates. The application then retrieves those fragments—again requesting H264 compression—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 the two fragments (m_1, m_2) as illustrated in Figure 3(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 3(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 3(b) shows the different selections that VSS might make to answer this read. Each *physical video fragment* $\{f_1, ..f_6\}$ in Figure 3(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* $c_t(f)$ that represents the cost of converting a physical video fragment f from a source physical format into a target physical format. The selected fragments must also 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., $u(m_0, f) \geq \tau$; see Section 3.2 a discussion of the quality model u). Our prototype sets a threshold $\tau = 40$dB, which is considered to be lossless (see Section 3.2). 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 [16]. 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 initial solution by continuing our example from Figure 3(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 3(a), the transition times include those in the set $\{30, 60, 70\}$, and we illustrate them in Figure 3(b) by partitioning the set of cached materialized views at each transition point. We also omit 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 3(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 3 requires transcoding from physical format $P =$ HEVC to $P' =$ H264 with no change in spatiotemporal format (i.e., $S = S'$).

During installation, VSS computes the domain of α by executing the vbench benchmark [33] on the installation hardware, which produces per-pixel transcode speedups for a variety of resolutions and codecs. For resolutions not evaluated by vbench, VSS approximates α by piecewise linear interpolation of the benchmarked resolutions.

We must also consider the data dependencies between frames. Consider the illustration in Figure 4, 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*, as shown in gray. 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.

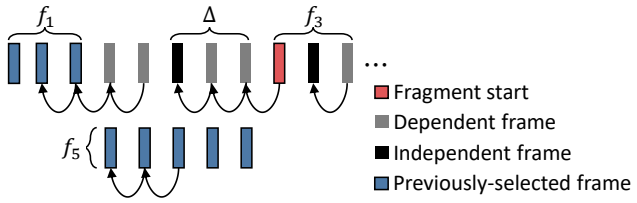


Figure 4: A simplified illustration based on the fragments in Figure 3. VSS is considering using fragment f_3 starting at the red-highlighted frame and has already decided to use f_1 and f_5. However, f_3 cannot be decoded without transitively decoding its dependencies shown by directed edges (labeled Δ). Delete this figure?

To model this cost, we introduce a *look-back cost* $c_l(P, f)$ that gives the cost of decoding the frames Δ in fragment f along with the dependent frames not part of f and *if they have not already been decoded*, meaning that they are not a member of the previously selected fragments P. As illustrated in Figure 4, 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 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 $c_l(P, f) = |A - P| + \eta \cdot |\Delta - A - P|$.

To conclude our example, observe that our goal is to choose a set of physical video fragments that (i) cover the queried spatiotemporal range, (ii) do not temporally overlap, and (iii) minimize the decode and look-back cost of selected fragments. In Figure 3, of all the possible paths between 20 and 80, the one with the lowest cost—highlighted in Figure 3(b)—minimizes the total cost of producing the answer. These characteristics collectively meet the requirements identified at the beginning of this section.

Specifically, 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 by adopting 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 answering 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 ($\epsilon = 35$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 n pixels in frame f_{i_j} relative to the corresponding pixels in reference frame f_{0_j}, normalized by the maximum possible pixel value I (generally 255):

$$PSNR(f_i, f_0) = \frac{1}{n}\sum_{j=0}^{n} 10 \cdot \log_{10}\left(\frac{I^2}{MSE(f_{i_j}, f_{0_j})}\right)$$

As described previously, error in f 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 re-decompressing f_0—which would be extremely expensive—VSS instead bounds $MSE(f_0, f_2)$ in terms of $MSE(f_0, f_1)$ and $MSE(f_1, f_2)$, which are a single real-valued aggregates stored as metadata. This bound is derived as follows for fragments of resolution $m \times n$:

$$MSE(f_0, f_2) = \frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(f_0^{ij} - f_2^{ij})^2$$

$$= \frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\left[(f_0^{ij} - f_2^{ij})^2 + 2(f_1^{ij})^2 - 2(f_1^{ij})^2\right.$$

$$
\begin{aligned}
&\quad + 2f_1^{ij} f_2^{ij} - 2f_1^{ij} f_2^{ij} + 2f_0^{ij} f_1^{ij} - 2f_0^{ij} f_1^{ij}\big] \\
&= \frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\big[(f_0^{ij} - f_1^{ij})^2 + (f_1^{ij} - f_2^{ij})^2 \\
&\qquad + 2f_1^{ij}(f_2^{ij} - f_1^{ij}) - 2f_0^{ij}(f_2^{ij} - f_1^{ij})\big] \\
&= MSE(f_0, f_1) + MSE(f_1, f_2) \\
&\quad + \frac{2}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\big[(f_1^{ij} - f_0^{ij}) \cdot (f_2^{ij} - f_1^{ij})\big] \\
&= MSE(f_0, f_1) + MSE(f_1, f_2) \\
&\quad + \frac{2}{mn}\Bigg[\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(f_1^{ij} - f_0^{ij}) \cdot (f_2^{ij} - f_1^{ij}) \cdot I(\cdot > 0) + \\
&\qquad \sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(f_1^{ij} - f_0^{ij}) \cdot (f_2^{ij} - f_1^{ij}) \cdot I(\cdot < 0)\Bigg] \\
&\leq MSE(f_0, f_1) + MSE(f_1, f_2) \\
&\quad + \frac{2}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(f_1^{ij} - f_0^{ij}) \cdot (f_2^{ij} - f_1^{ij}) \cdot I(\cdot > 0) \\
&\leq MSE(f_0, f_1) + MSE(f_1, f_2) \\
&\quad + \frac{2}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\left[\frac{(f_1^{ij} - f_0^{ij}) + (f_2^{ij} - f_1^{ij})}{2}\right]^2 \\
&= MSE(f_0, f_1) + MSE(f_1, f_2) \\
&\quad + \frac{1}{2mn}\Bigg[\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(f_2^{ij} - f_0^{ij})^2 \cdot I(\cdot > 0)\Bigg] \\
&\leq MSE(f_0, f_1) + MSE(f_1, f_2) + \frac{1}{2}MSE(f_0, f_2) \\
&= 2(MSE(f_0, f_1) + MSE(f_1, f_2))
\end{aligned}
$$

Using the above formulation, VSS efficiently estimates MSE for a sequence of two transformations without requiring that f_0 be available. Extension to transitive sequences is straightforward.

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 [33], a benchmark for evaluating video transcode performance in the cloud. To improve on this estimate, VSS periodically resamples regions of compressed video, computes exact PSNR, and updates its estimate.

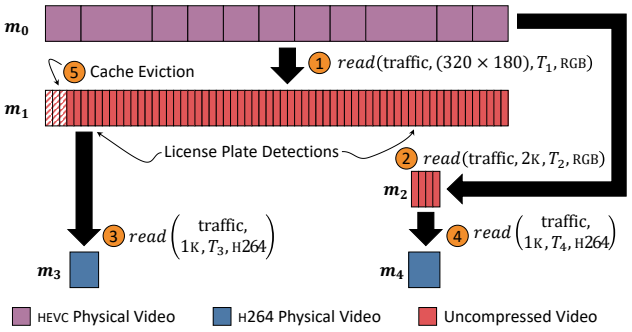


Figure 5: VSS caches read results and uses them to answer future queries. In ❶ an application reads `traffic` at 320×180 resolution for use in object detection, which VSS caches as m_1. In ❷ VSS caches m_2, a region with a dubious detection. In ❸ and ❹ VSS caches H264-encoded m_3 & m_4, where objects were detected. However, reading m_4 exceeds the storage budget and VSS evicts the striped region at ❺. Delete this figure?

4 DATA CACHING IN VSS

In the previous sections we described how VSS persists, reads, and writes physical videos. We now describe how VSS decides *which* physical videos to maintain, and which to evict under low disk space conditions. The caching process 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 of these 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 consisting of fragments with sufficiently high quality (PSNR ≥ 40dB in our prototype, which is considered to be lossless) relative to the originally ingested video.

As a running example, consider the sequence of reads illustrated in Figure 5, which mirrors the alert application described in Section 2. In this example, an application reads a low-resolution uncompressed video from VSS for use with an automobile detection algorithm. VSS caches the result as a sequence of three-frame GOPs (approximately 518kB per

GOP). One detection was marginal, and so the application reads higher-quality 2K video to apply a more accurate detection model. VSS caches this result as a sequence of single-frame GOPs, since each 2K RGB frame is 6MB in size. Finally, the application extracts two H264-encoded regions for offline viewing. VSS caches m_3, but when executing the last read it determines that it has exceeded its storage budget and must now decide whether to cache m_4.

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 are forced to make decisions on large, indivisible, and opaque entries (a far more challenging problem space with limited progress in formulating approximate solutions [8]).

However, there are several key differences between GOPs and pages. In particular, GOPs are related to each other:

- One GOP might be a higher-quality or version of another.
- Consecutive GOPs form a contiguous video fragment.

These correlations make typical eviction policies like least-recently used (LRU) problematic. In particular, application of 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; see Section 3).

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 modified as follows:

- **Position (p).** We increase the sequence number of fragments occurring near the middle of a physical video, relative to those at the beginning or end. Given a physical 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).** We decrease the sequence number of fragments that have redundant or higher-quality variants. To do so, using our 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 $r(f_i)\colon \mathbb{Z}^{0+}$ 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).** We never evict 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, we increase the sequence number of each fragment by:

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

  As we discuss in Section 3, our prototype sets $\tau = 40$.

Using the weights described above, VSS modifies the sequence number of each candidate fragment f_i by $LRU_{\text{vss}}(f_i) = LRU(f_i) + \gamma \cdot p(f_i) - \zeta \cdot r(f_i) + b(f_i)$. 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.

In Figure 5, we show application of LRU_{vss} where VSS choses to evict the three-frame GOP at the beginning of m_1 and to cache m_4. If our prototype had instead weighed $\zeta \gg \gamma$, VSS would elect to evict m_3 since it was not recently used and is the variant with lowest quality.

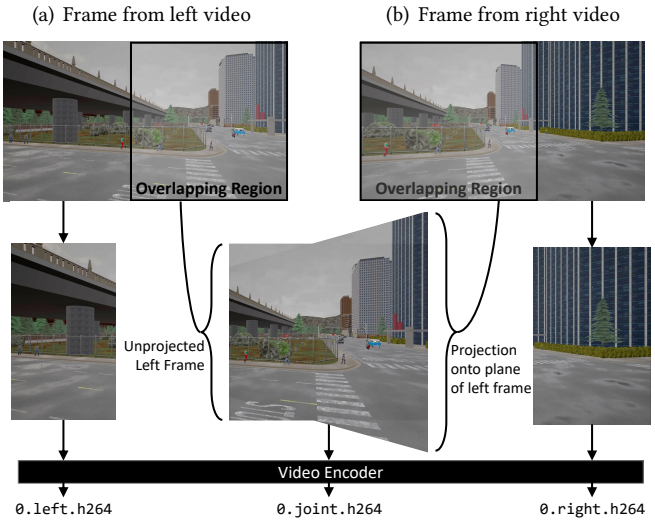
5 DATA COMPRESSION IN VSS

As described in Section 2, when an application writes data to VSS, VSS partitions the written video into blocks by GOP (for compressed video data) or contiguous frames (for uncompressed video data). VSS follows the same process when caching the result of a read operation for future use.

To improve the storage performance of written and cached video data, 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 performs compression on blocks of uncompressed, infrequently-accessed GOPs (Section 5.2); and (iii) improves the performance of reads by compacting temporally-adjacent physical videos (Section 5.3).

5.1 Joint Physical Video Compression

Increasingly large amounts of video content is produced from cameras that are spatially proximate with similar orientations. For example, a bank of traffic cameras mounted on a pole will each capture video of the same intersection from similar angles. Although the amount of "overlapping video" being produced is difficult to quantify, it broadly includes traffic cameras (7.5PB per day in the United



(a) Frame from left video (b) Frame from right video

(c) Left, overlapped, and right regions are separately encoded.

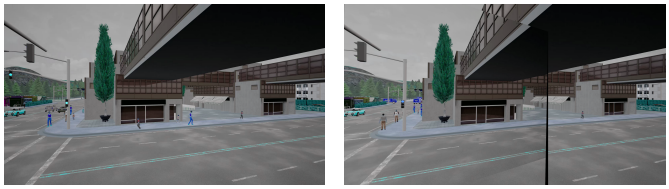
Figure 6: Joint compression applied to two horizontally-overlapping frames. VSS identifies overlapping regions in each frame, combines them, and separately encodes each piece (left, overlap, & right).

(a) Recovered frame from left video (b) Recovered frame from right video

Figure 7: Recovered frames from joint compression.

Kingdom [9]), body-worn cameras (>1TB per day [59]), autonomous vehicles (>15TB per vehicle per hour [21]), along with videos of tourist locations, concerts, and political events.

Despite the redundant information that mutually exists in these video streams, most applications treat these video streams as distinct and persist them separately to disk. VSS optimizes the storage of these videos by reducing the redundancy between pairs of highly-similar video streams. We apply this *joint compression* optimization between pairs of GOPs in different logical videos. As first step, VSS finds candidate GOPs to jointly compress. Then, given a pair of overlapping GOP candidates, VSS recompresses them frame-by-frame. We describe how two frames are compressed in Section 5.1.1. For static cameras, once VSS compresses the first frame in a GOP, it can reuse the information it has computed to easily compress subsequent frames in the same GOP. We describe joint compression for dynamic cameras in Section 5.1.2. We finally describe the search process for overlapping GOPs in Section 5.1.3.



(a) Original frame (b) Recovered frame

Figure 8: An example recovered frame with slightly incorrect homography. In this case VSS detects the error relative to the original frame and aborts joint compression.

5.1.1 Joint frame compression. Figure 6 illustrates the joint compression process for two frames taken from a synthetic dataset (Visual Road-1K-50%, described in Section 6). Figure 6(a) and Figure 6(b) respectively show the two frames with the overlapping regions highlighted, and Figure 6(c) shows the overlapping regions combined.

Because these frames were captured from cameras at different orientations, combining them is non-trivial and requires more than an isomorphic translation or rotation (e.g., the angle of the horizontal sidewalk is not aligned in the two frames). Instead, VSS estimates a homography between the two frames and the resulting projection is used to transform between the two spaces. As shown in Figure 6(c), VSS applies this transformation to the right frame which causes its right side to bulge vertically. However, after it is overlaid onto the left frame, the two align near-perfectly.

As formalized in Algorithm 1, joint projection proceeds as follows. First, VSS estimates a homography between the two frames. To do so, VSS reads two frames from the GOP pairs being compressed. Next, it applies a feature detection algorithm [34] that identifies similar features that co-occur in both frames. Using these features and random sample consensus (RANSAC; [14]), it next estimates the homography matrix used to transform between frame spaces.

With the homography estimated, VSS uses it to project the right frame into the space of the left frame. This results in three distinct regions: (i) a "left" region of the left frame that does not overlap with the right, (ii) an overlapping region, and (iii) a "right" region of the right frame that does not overlap with the left. VSS splits these into three distinct regions and uses an ordinary video codec to encode each region separately and write it to disk.

When constructing the overlapping region, VSS applies a *merge function* that transforms overlapping pixels from each overlapping region and outputs a merged, overlapping frame. An *unprojected merge* favors the unprojected frame (i.e., the left frame in Figure 6(c)), while a *mean merge* averages the pixels from both input frames.

Algorithm 1 Joint compression algorithm

let HOMOGRAPHY(f, g) estimate the 3×3 homography matrix of f and g
let u, τ, ϵ respectively be the VSS quality model (Section 3.2), quality threshold (Section 3.1), and duplicate threshold (Section 5.1.1).

function JOINT-COMPRESS(F, G, m)
  Input: Video frames $F = \{f_1, \ldots, f_n\}$
  Input: Video frames $G = \{g_1, \ldots, g_n\}$
  Input: Merge function m
  Output: Vector of compressed subframes

  $H, C, i, j \leftarrow$ HOMOGRAPHY(f_1, g_1), $\emptyset$, 0, 0
  if $H = \emptyset$ **then return** $\emptyset$ ▷ No homography found
  else if $H_{1,2} < 0$ **then return** JOINT-COMPRESS(G, F) ▷ Reverse transform

  while $i \leq n$ **do**
    if $||H - \mathcal{I}||_2 \leq \epsilon$ **then**
      $S_i, H \leftarrow (\emptyset, f_1, \emptyset), \mathcal{I}$ ▷ Duplicate Frames
    else
      $S_i \leftarrow$ PARTITION(f_i, g_i, H, m)
    if $S_i = \emptyset \vee u\left(f_i, \left[S_i^{\text{left}}\ S_i^{\text{overlap}}\right]\right) < \tau \vee$ ▷ Verify quality
      $u\left(g_i, \left[\text{TRANSFORM}(S_i^{\text{overlap}}, H^{-1})\ S_i^{\text{right}}\right]\right) < \tau$ **then**
      if j = 0 **then** ▷ Recompute homography
        $H, j \leftarrow$ HOMOGRAPHY(f_i, g_i), $j + 1$
      else
        return $\emptyset$ ▷ Abort joint compression
    else
      $C \leftarrow C \oplus$ COMPRESS(S_i)
      $i, j \leftarrow i + 1, 0$
  return C

function PARTITION(f, g, H, m)
  Input: Video frame f with size n, m
  Input: Video frame g with size n, m
  Output: Left, overlap, and right subframes

  $x_f \leftarrow \left[H^{-1} \cdot \begin{bmatrix}0 & 0 & 1\end{bmatrix}^\intercal\right]_2$
  $x_g \leftarrow n - \left[H \cdot \begin{bmatrix}n & 0 & 1\end{bmatrix}^\intercal\right]_2$
  if $\neg(0 < x_f \leq n) \vee \neg(0 < x_g \leq n)$ **then return** $\emptyset$

  $l, r \leftarrow f[1, x_f], g[x_g, n]$ ▷ Left and right subframes
  $o \leftarrow m(f[x_f, n],$ TRANSFORM$(g[1, x_g], H))$ ▷ Overlap of f and g
  return (l, o, r)

function TRANSFORM(f, H) ▷ Compute perspective transform of f
  return $\begin{bmatrix} f'_{1,1} & \cdots \\ \vdots & \ddots \end{bmatrix}$, where $t_{i,j} = H \cdot \begin{bmatrix} i \\ j \\ 1 \end{bmatrix}$, $f'_{i,j} = f_{\frac{[t_{i,j}]_1}{[t_{i,j}]_3}, \frac{[t_{i,j}]_2}{[t_{i,j}]_3}}$

During reads, VSS reverses this process to produce the original frames. Figure 7 shows two such recovered frames produced using the frames shown in Figure 6.

Some frames stored in VSS may be exact duplicates, however, for which the projection process described above introduces unnecessary computational overhead. VSS detects this case by checking whether the homography matrix would make a near-identity transform (specifically by checking $||H - \mathcal{I}||_2 \leq \epsilon$, where $\epsilon = \frac{1}{10}$ in our prototype). When this condition is met, VSS instead replaces the redundant GOP with a pointer to its near-identical counterpart.

5.1.2 Dynamic & mixed resolution cameras. For stationary and static cameras, the originally-computed homography is sufficient to jointly compress all frames in a GOP. For dynamic cameras, however, the homography quickly becomes outdated and, in the worst case, the cameras may no longer overlap. To guard against this, VSS adopts the following approach. For each jointly compressed frame, VSS inverts the projection process and recovers the original frame. It then compares the recovered variant against the original frame using its quality model (see Section 3.2). If the quality has dropped below some threshold (24dB in our prototype), VSS re-estimates the homography and reattempts joint compression. If the reattempt is also of insufficient quality, VSS aborts joint compression.

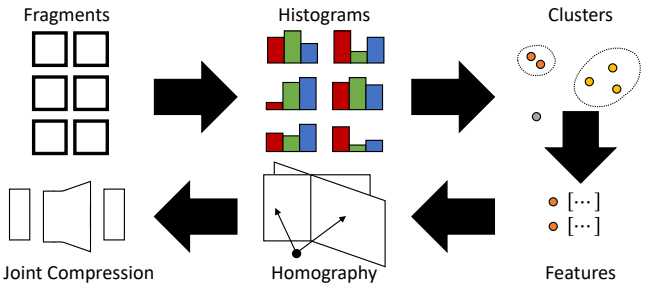


Figure 9: VSS's joint compression fragment selection process begins by computing and clustering histograms of ingested fragments. For the smallest cluster, it computes features and searches for other fragments that have large number of similar features. For each pair, VSS computes homography and jointly compresses. Delete this figure?

For both static and dynamic cameras, VSS may occasionally poorly estimate the homography between two otherwise-compatible frames. The recovery process described above also identifies these cases. When detected (and if re-estimation is unsuccessful), VSS aborts joint compression for that pair of GOPs. An example of two frames where VSS produced an incorrect homography is illustrated in Figure 8.

VSS may also identify joint compression candidates that are at dissimilar resolutions. To handle this case, VSS first upscales the lower resolution fragment to that of the higher. It then applies joint compression as usual, again confirming that the resulting merged fragments are of sufficient quality.

5.1.3 Selecting GOPs for joint compression. Thus far we have discussed how VSS applies joint compression to a pair of GOPs, but not how the pairs are selected. Since the brute force approach of evaluating all $O(n^2)$ pairs is prohibitively expensive, VSS instead uses the multi-step process illustrated in Figure 9. First, to reduce the search space, VSS clusters all video fragments using their color histograms. Videos with highly distinct color histograms are unlikely to benefit from joint compression. The VSS prototype implementation uses the BIRCH clustering algorithm [61], which is memory efficient, scales to many data points, and allows VSS to incrementally update its clusters as new GOPs arrive.

Once VSS has clustered the ingested GOPs, it selects the cluster with the smallest radius and considers its constituents for joint compression. To do so, VSS applies a modified form of the homography computation described above. It begins by applying the feature detection algorithm [34] from Section 5.1.1. Each feature is a spatial histogram characterizing an "interesting region" in the frame (i.e., a keypoint).

VSS next looks for other GOPs in the cluster that share a large number of interesting regions. Thus, for each GOP, VSS iteratively searches for similar features (i.e., within distance d) located in other GOPs within the cluster. A correspondence, however, may be ambiguous (e.g., if a feature in GOP 1 matches to multiple, nearby features in GOP 2). VSS rejects such matches.

When VSS finds m or more nearby, unambiguous correspondences, it considers the pair of GOPs to be sufficiently related. It then applies joint compression to the GOP pair as described above. Note that the joint compression algorithm described in Section 5.1.1 may abort if joint compressing the GOPs does not produce a sufficiently high-quality result (i.e., a false positive). Our prototype sets $m = 20$, requires features to be within $d = 400$ (using a Euclidean metric), and disambiguates using Lowe's ratio [35].

5.2 Deferred Compression

Most video-oriented applications operate over decoded video data (e.g., RGB). Such data is vastly larger than its compressed counterpart: storage for a typical 4K video exceeds five terabytes per hour when uncompressed (e.g., the VisualRoad-4K-30% dataset we describe in Section 6 is 5.2TB uncompressed as 8-bit RGB). As VSS caches uncompressed video regions as the result of reads (e.g., while scanning for license plates), it quickly exhausts the video's storage budget.

To mitigate this, VSS adopts the following approach. When a video's cache size exceeds some threshold (25% in our prototype), VSS activates a special *deferred compression* mode. In this mode, when a read requests uncompressed data, VSS examines the current cache and orders the uncompressed physical video entries by eviction order. It then losslessly compresses the *last* entry in this list (i.e., the entry least likely to be evicted). It then executes the read as usual.

Our prototype uses Zstandard for lossless compression, which emphasizes compression and decompression speed but has a lower compression ratio relative to more expensive image and video codecs such as PNG and HEVC [13].

VSS performs two additional optimizations beyond the approach described above. First, Zstandard comes with a "compression level" setting, which is an integer in the range [1..19], with the lowest setting having the fastest speed but the lowest compression ratio (and the highest setting having the opposite characteristics). VSS linearly scales Zstandard compression level with the remaining storage budget, which has the effect of decreasing compressed size while increasing compression time. Second, while deferred compression is active, VSS continues to compress cache entries in a background thread during periods when no other IO requests are being executed.

5.3 Physical Video Compaction

As a result of caching the result of user queries, VSS may persist pairs of cached videos that contain data in contiguous time and with the same spatial and physical configurations. For example, the cached reads resulting over a logical video at time [0, 90] and [90, 120] are contiguous. Application of lazy compression may also create contiguous physical videos. For example, if VSS lazily compressed an uncompressed physical video starting at time 120, it would be contiguous with the cached video covering time [90, 120].

To reduce the number of physical videos that need to be considered in performing a read, VSS periodically compacts pairs of contiguous cached videos and substitutes a unified representation. To do so non-quiescently, it periodically examines pairs of cached videos and, for each contiguous pair, uses hard links to merge the GOPs from the second into the first. The result is a unified cached video that contains the aggregated video data from both sources.

6 EVALUATION

We have implemented a prototype of VSS using approximately 4,000 lines of C++ and Python code. GPU-based operators were implemented using CUDA [42], NVENCODE/NVDECODE [41], and OpenCV [43]. VSS also uses FFmpeg [7] for video plumbing operations such as GOP decoding, segmentation, concatenation, and SQLite [48] for metadata storage. Our prototype currently adopts a no-overwrite policy for logical videos and disallows updates. We plan on supporting both features in a future release. Finally, when performing writes, VSS does not guarantee that data are visible to readers until the file being written is closed.

We evaluate VSS in terms of read (Section 6.1), write and caching (6.2), and compression (6.3) performance.

Baseline systems. We compare VSS against VStore [58], a recent storage system that supports video analytic workloads by pre-computing multiple video representations. We also evaluate against direct use of the local file system.

We build VStore with support for GPU-accelerated video encoding and decoding, and where available utilize these accelerated operations. We experienced intermittent failures when running VStore on >2,000 frame videos, and to work around this limit all VStore experiments to this size.

Table 1: Datasets used to evaluate VSS

Dataset	Resolution	# Frames	Compressed Size (MB)
Robotcar	1280×960	7,494	120
Waymo	1920×1280	398	7
VisualRoad 1K-30%	960×540	108k	224
VisualRoad 1K-50%	960×540	108k	232
VisualRoad 1K-75%	960×540	108k	226
VisualRoad 2K-30%	1920×1080	108k	818
VisualRoad 4K-30%	3840×2160	108k	5,500

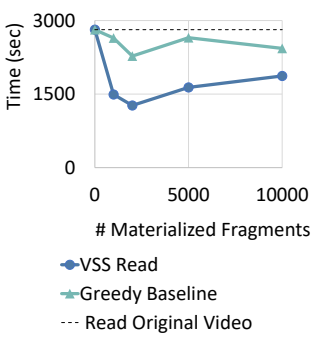


Figure 10: Time to select fragments and read video.

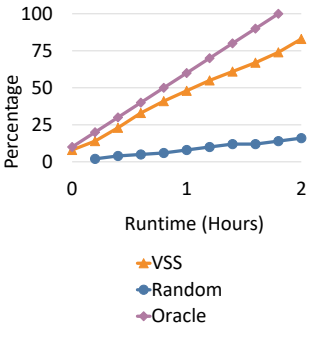


Figure 11: Joint compression pair selection.

Experimental configuration. We perform all experiments using a single-node system equipped with an Intel i7-6800K processor with 6 cores running at 3.4Ghz and 32GB DDR4 RAM. The system also includes a Nvidia P5000 GPU with two discrete NVENCODE chipsets.

Datasets. In our evaluation, we use a combination of real and synthetic video data. We use the former to measure VSS performance under real-world inputs, while the latter allows us to test on a variety of carefully-controlled configurations. We use the datasets shown in Table 1 for the experiments throughout this section. The "Robotcar" dataset consists of two highly-overlapping videos captured using adjacent stereo cameras mounted on a moving vehicle [38]. The dataset is provided as 7,494 separately-compressed frames recorded at 30 FPS and compressed using the PNG codec (as is common for datasets that target machine learning). We transcoded these frames into an H264 video with one-second GOPs.

The "Waymo" dataset is an autonomous driving dataset [54]. We selected one segment (approximately twenty seconds) from the dataset, which was captured using two vehicle-mounted cameras. Unlike the Robotcar dataset, we estimate that Waymo videos overlap by approximately 15%.

Finally, the various "VisualRoad" datasets consist of synthetic video generated using a recent video analytics benchmark designed to evaluate the performance of video-oriented data management systems [20]. To generate each dataset, we used Visual Road to generate a one-hour simulation and produce video data at 1K, 2K, and 4K resolutions. We also modified the field of view of each panoramic camera in the simulation so that we could vary the horizontal overlap of the resulting videos. We repeated this process several times and produced five distinct datasets; for example, the "VisualRoad-1K-75%" dataset contains two one-hour videos with 75% horizontal overlap.

Because the size of the uncompressed 4K Visual Road dataset (~ 5TB) exceeds our storage capacity, we do not show results that require fully persisting this dataset uncompressed on disk.

6.1 Data Retrieval Performance

Long Read Performance. Our first experiment explores the read performance of VSSfor large reads using various numbers of physical videos generated by cached reads. In this experiment, we vary the number and types of fragments available in the cache. First, repeatedly execute queries of the form *read*(VisualRoad-4K-30%, 3840×2160, $[t_1, t_2]$, P), with times $0 \le t_1 < t_1 + 1 \le t_2 < 3600$ (in seconds) along with a physical format $P \in \{$H264, HEVC, RGB, YUV420, YUV422, NV12$\}$, with t_1, t_2, and P drawn uniformly at random. These cache entries might be generated by the application of a machine learning algorithm (e.g., license plate detection) over many regions of a video. We iterate this process until VSS has cached a variable number of physical videos. For this experiment we assume an infinite storage budget.

We then execute a maximal read (i.e., from time 0 to 3600 seconds) of this dataset in the HEVC format, which is different from the format of the originally-written physical video (H264) and allows VSS to leverage its materialized fragments.

We show the performance of this read in Figure 10 for the application scenario. Since none of the other baseline systems support automatic conversion from H264 to HEVC, we do not show their runtimes for this experiment.

As we see in Figure 10, even a cache with a small number of entries can improve read performance by a substantial amount—28% at 100 entries and up to a maximum improvement of 54%. We further observe that utilizing VSS's solver-based fragment selection algorithm outperforms both reading the original video and a naïve baseline that greedily selects fragments. This is because VSS decodes fewer redundant dependent frames.

Short Read Performance. We next examine VSS performance when reading small, one-second regions of video (e.g., to apply license plate detection only to regions of video that contain automobiles). In this experiment, we begin with the VSS state generated by the previous experiment and execute many short reads of the form

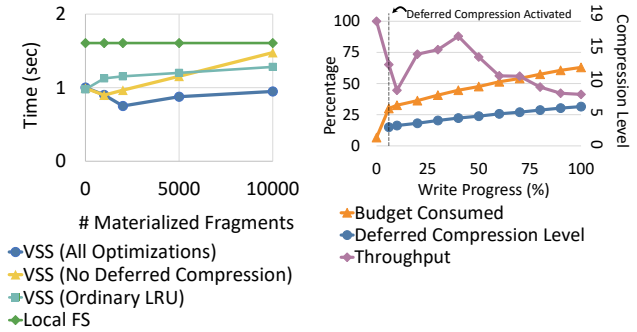
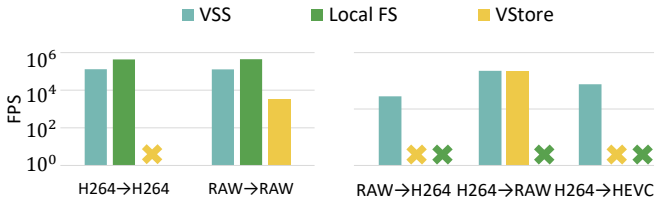


Figure 12: Selecting and reading short segments.

Figure 13: Writes with deferred compression.



(a) Read in same format (b) Read in different format

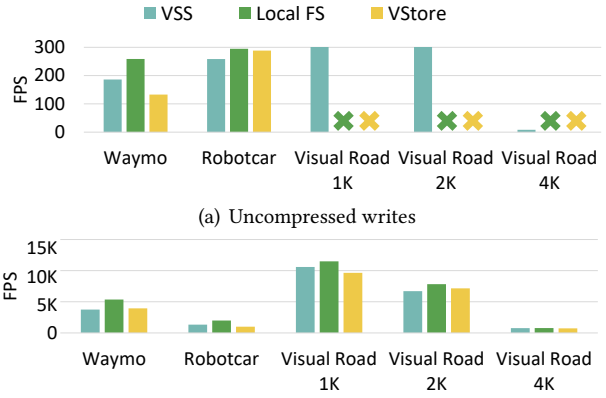
Figure 14: Read throughput for VSS and baseline systems. Each group $I \rightarrow O$ shows throughput reading video in format I and outputting it in format O. An × means that a system does not support the read type.

read(VisualRoad-4K-30%, R, $[t_1, t_2]$, P), where $0 \leq t_1 < 3600$ and $t_2 = t_1 + 1$ (i.e., random 1 second sequences), $R \in \{1\text{K}, 2\text{K}, 4\text{K}\}$ and P is as in the previous experiment.

Figure 12 shows the result for VSS ("VSS (All Optimizations)") versus reading from the original video from the local file system ("Local FS"). In this experiment, VSS is able to offer improved performance due to its ability to serve from a cache of lower-cost fragments, rather than transcoding the source video.

Read Format Flexibility. Our next experiment evaluates VSS's ability to transparently read video data in a variety of formats. To evaluate this functionality relative to the baseline systems, we first write the VisualRoad-1K-30% dataset to VSS, VStore, and the local disk. We write the data to each system in both compressed (224MB) and uncompressed form (approximately 328GB).

We use an empty cache for VSS and read the persisted videos from each system in various formats and measure the throughput offered by each system. Figure 14 shows results for a read in the same format (Figure 14(a)) written to a file system and different formats (Figure 14(b)). Because the local file system does not support automatic representation transformation (e.g., converting H264-compressed video into RGB), we do not show results for these cases. Additionally, VStore does not support reading some formats from its store, and we additionally omit its result for this case.



(a) Uncompressed writes

(b) Compressed writes

Figure 15: Throughput to write uncompressed and H264 compressed data to VSS and baseline systems. An × means lack of support for the write type.

We find that read performance *without* a format conversion from VSS is modestly slower than the local file system, due in part to the local file system being able to execute entirely without kernel transitions and the need for VSS to concatenate many individual GOPs. However, our results show that VSS can adapt to reads in *any* format, a benefit not available when using the local file system.

We additionally find that VSS performance outperforms VStore when reading uncompressed video and is similar when transcoding H264. At the same time, VSS offers more flexible input and output format options and does not require a workload to be specified in advance.

6.2 Data Persistence & Caching

Write Throughput. Our next evaluation explores VSS write and caching performance. To evaluate the write performance of VSS relative to the other baseline systems, we write each dataset to the respective systems in both compressed and uncompressed form. We measure the write throughput of each system and report the results in Figure 15(a).

For datasets that will fit on local storage, VSS performs similarly to using the local file system and VStore, though VSS outperforms VStore for the extremely small Waymo dataset. On the other hand, none of the baseline systems have the capacity to store the larger uncompressed datasets. For example, the uncompressed VisualRoad-4K-30% dataset is over five terabytes. However, as the video's storage budget reaches capacity, VSS can activate deferred decompression and automatically compress the data being written. This allows it to store datasets that no other system can handle (at the cost of decreased throughput).

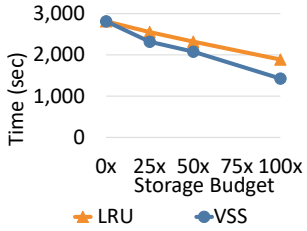


Figure 16: Read runtime by cache eviction policy.

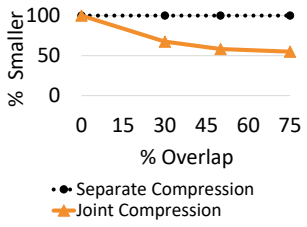
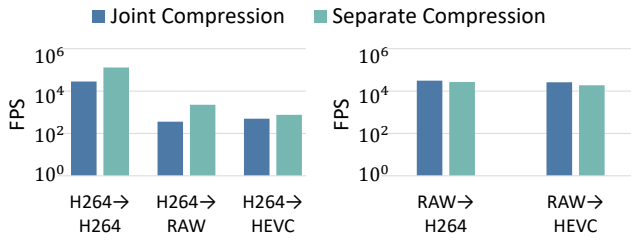


Figure 17: Joint compressed vs separate video.



(a) Read throughput (b) Write throughput

Figure 18: Throughput for writes with and without the joint compression optimization.

Table 2: Joint compression recovered quality

Dataset	Quality (PSNR) Left	Quality (PSNR) Right	Fragments Admitted (%)
Unprojected merge			
Robotcar	350	24	36
Waymo	352	29	39
VRoad-1K-30%	359	30	46
VRoad-1K-50%	358	28	41
VRoad-1K-75%	348	24	44
VRoad-2K-30%	352	30	52
VRoad-4K-30%	360	30	54
Mean merge			
Robotcar	30	27	64
Waymo	32	30	68
VRoad-1K-30%	31	30	80
VRoad-1K-50%	29	29	72
VRoad-1K-75%	30	28	68
VRoad-2K-30%	30	30	82
VRoad-4K-30%	29	30	78

We next write the compressed evaluation datasets to each store. Figure 15(b) shows the performance results for each baseline system. Here all systems perform approximately equivalently, with both VSS and VStore exhibiting minor overhead relative to the local file system.

Cache Performance. To evaluate the VSS cache eviction policy, we repeat our experimental setup for read performance in Section 6.1. We execute 5,000 random read operations to populate the VSS cache. However, instead of assuming an infinite storage budget, we limit it to be various multiples of the input size (e.g., 25×) and apply either the least-recently used (LRU) or VSS eviction policy. This limits the number of physical videos available for performing reads.

With the cache populated, we execute a final maximal read for the entire video range (i.e., [0, 3600]). Figure 16 shows the runtimes for each policy and storage budget. This shows that VSS can reduce read execution relative to LRU.

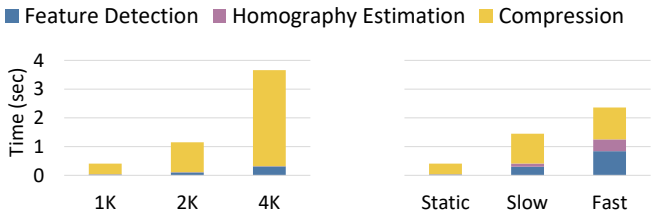
6.3 Compression Performance

Joint Compression Quality. In this evaluation we examine the recovered quality of jointly-compressed physical videos. For this experiment we write various overlapping Visual Road datasets to VSS. We then subsequently read each video back from VSS and compare the resulting quality against its originally-written counterpart. We use the peak signal-to-noise ratio (PSNR) as a quality comparison metric.

Table 2 gives the PSNR for recovered data compared against the written videos. Recall that a PSNR of ≥40 is considered to be lossless, and ≥30 near-lossless [22]. When applying the unprojected merge function during joint compression, we find almost perfect recovery for the left input (with PSNR values exceeding 300dB), and near-lossless quality for the right input. Loss in fidelity occurs when inverting the merge, i.e., performing the inverse projection on the right frame using left-frame pixels decreases the quality of the recovered frame. This merge function also leads to VSS rejecting approximately half of the fragments due to their falling below the minimum quality threshold. We conclude this merge function is useful for reducing storage size in video data that must maintain at least one perspective in high fidelity.

On the other hand, we find balanced, near-lossless quality for *both* the left and right frames when applying the mean merge function during joint compression. Additionally, the number of fragments admitted by the quality model is substantially higher under this merge function. Accordingly, the mean merging function is appropriate for scenarios where storage size is paramount and near-lossless degradation is acceptable.

Joint Compression Throughput. Our next experiment examines the VSS read throughput with and without the joint compression optimization applied. For this experiment, we write each video in the VisualRoad-1K-30% dataset to VSS, once with joint compression enabled and separately with it disabled. We then execute a read operation in various physical configurations and for the entire duration. Figure 18(a) shows the throughput achieved when executing



(a) By resolution (b) By dynamicism

Figure 19: Joint compression throughput by resolution (a) and by camera dynamicism (b). For dynamic cameras, "slow" requires homography recalculation on 1K video every fifteen frames and "fast" every five frames.

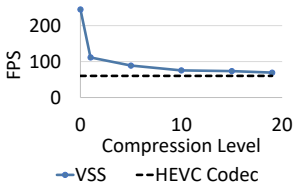


Figure 20: Throughput for reads over fragments with deferred compression.

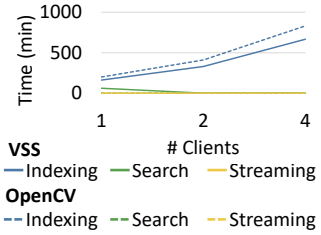


Figure 21: End-to-end application performance.

the read using each configuration. Our results indicate that the read overhead for videos stored using joint compression is modest but similar to reads that are not co-compressed.

Applying joint compression to a pair of videos requires several nontrivial operations, and our next experiment evaluates the overhead associated with its execution. For this experiment, we write 1K, 2K, and 4K video and measure throughput. Figure 18(b) shows the results of this experiment. Surprisingly, joint writes are similar to writing each video stream separately. This speedup is due to VSS's encoding of each of the three lower-resolution streams in parallel, and since compression time is roughly proportional to resolution, encoding the three lower-resolution components is faster than the original frame. Additionally, the overhead in feature detection and generating homography does not obviate this performance advantage. Figure 19 decomposes joint compression overhead into these subcomponents. First, Figure 19(a) measures joint compression overhead by resolution, where we can see that compression dominates. Figure 19(b) further shows VSS performance under three additional scenarios: a static camera, a slowly rotating camera that requires homography reestimation every fifteen frames, and a rapidly rotating camera that requires reestimation every five frames. In these scenarios non-compression costs scale with reestimation period, and compression performance is loosely correlated since a new keyframe is needed after every homography change.

Our final experiment evaluates the performance of VSS's joint compression selection algorithm. We write the videos listed in Table 1 into VSS and record the number of joint compression candidates that VSS identifies and merges. We show three strategies: VSS's algorithm (see Section 5), one that uses an oracle to select the correct pairs, and random selection.

Figure 11 illustrates the performance of each strategy. VSS's clustering approach can identify 84% of the total applicable pairs in time similar to the oracle-based approach. Both strategies outperform uniform random sampling.

Joint Compression Storage. To show the storage benefit of VSS's joint compression optimization, we separately applied the optimization to each of the Visual Road videos. We then measured the final on-disk size of the resulting videos against their separately-encoded variants. Figure 17 shows the result of this experiment. As we can see, joint compression substantially reduces the storage requirements of overlapping video.

Deferred Compression Performance. We next evaluate the performance of deferred compression for uncompressed writes by storing 3600 frames of the VisualRoad-1K-30% dataset in VSS, leaving storage budget and deferred compression at their defaults. We periodically extract the storage used, Zstandard level, and write throughput.

The results for these metrics are listed in Figure 13. We show storage used as a percentage of the budget. Similarly, we show throughput relative to writing without deferred compression activated. Finally, we show compression level as a value in the interval [1..19]. As expected, storage used exceeds the deferred compression threshold early in the write, and a slope change shows that deferred compression is having a moderating influence on write size. Compression level scales linearly with storage used. Finally, throughput drops substantially as compression is activated, recovers considerably, and then slowly degrades as the compression level is increased.

Similarly, Figure 20 shows throughput for *reading* fragments of raw video compressed at various levels. Though reads for these fragments have additional overhead relative to uncompressed reads (and increased variance), reads continue to be faster than via traditional video codecs.

Finally, Figure 12 explores the trade-offs between deferred compression performance and VSS's cache eviction policy. In this experiment we variously disabled deferred compression ("VSS (No Deferred Compression)") and modified VSS to use ordinary LRU ("VSS (Ordinary LRU)"). The results show that VSS benefits from its eviction policy for small numbers of fragments (when deferred compression is off or at a low level), but offers increasingly large benefits as the cache grows.

6.4 End-to-End Application Performance

Our final experiment evaluates the performance of the end-to-end application described in Section 2. In this scenario, VSS serves as the storage manager for an application monitoring an intersection for automobiles associated with adults with dementia. Its execution involves three steps: (i) an *indexing phase* that identifies video frames containing an automobiles using a machine learning algorithm, (ii) a *search phase* that, given an alert for a missing vehicle, queries VSS for video frames containing vehicles with matching colors, and (iii) a *content retrieval phase* that retrieves video clips of vehicles of a given color.

We implemented this application using VSS and a variant that reads video data using OpenCV from the local file system. For indexing, we identified automobiles using the YOLOv4 model (both variants leveraged OpenCV to perform inference using this model). For the search task, we identify vehicle color by cropping video to the detected bounding box and identifying the dominant color. We computed the dominant color by generating a color histogram of the cropped region and computing the Euclidean norm between the largest bin and the search color, and consider a successful detection to occur when this distance is ≤ 25. In the content retrieval phase, we generate n video clips by retrieving groups of contiguous frames containing automobiles of the search color.

We use as input four extended two-hour variants of the Visual Road 2K dataset. We simulated execution by $m \in \{1, 2, 4\}$ clients by, for both the VSS and OpenCV variants, launching each step m times as a separate process. We index automobiles every ten frames (i.e., three times a second). All steps exhaust all CPU resources at > 4 clients, and so we limit concurrent requests to this maximum.

Figure 21 shows performance of each application step. The indexing step is a CPU-intensive operation that necessitates both video decoding and model inference, and because VSS introduces low overhead for reads, both variants perform similarly. Conversely, VSS excels at executing the search step, which requires retrieving raw, uncompressed frames that were cached during the indexing step. As such, it substantially outperforms the OpenCV variant. Finally, VSS's ability to minimize cost by efficiently identifying the lowest-cost transcode solution enables it to execute the search step substantially faster than the OpenCV variant. We conclude that VSS's performance greatly improves end-to-end application performance for video queries that depend on cached video in multiple formats.

7 RELATED WORK

Increased interest in applied machine learning and computer vision has led to the development of a number of new and proposed systems that target video analytics, including Optasia [36], LightDB [19], VisualWorldDB [18], Chameleon [27], Panorama [62], Vaas [5], SurvQ [49], and Scanner [45]. These systems utilize a local or distributed file system that can be modified to leverage a storage manager like VSS. Video-oriented accelerators such as BlazeIt [29], VideoStorm [60], Focus [23], NoScope [30], Odin [50], SQV [56], MIRIS [4], Tahoma[3], and Deluceva [53] can also transparently benefit from VSS (both for training and inference).

While the systems community has a long history of introducing specialized storage systems (e.g., HDFS [47]), few recent systems target video analytics (although others have identified this need [15, 28]). VStore [58] is one example that targets machine learning workloads by staging video in pre-specified formats. However, VStore requires the developer to know beforehand the specific workload and does not take advantage of data independence to improve performance beyond preselected materializations. By contrast, quFiles exploits data independence at the granularity of entire videos [52]. Others have explored on-disk layout of video data in the context of scalable streaming [32]. Other related systems such as Haystack [6], AWS Serverless Image Handler [1], and VDMS [46] emphasize image and metadata operations.

Interest in edge processing and networked cameras in the context of video analytics is also emerging, prompting applications that exploit clusters of networked cameras (e.g., VideoEdge [2, 24, 26, 57]). Since these cameras have constrained storage and compute resources, they would benefit from a storage system such as VSS that can transparently balance these factors and improve performance.

Techniques similar to VSS's joint compression optimization have been explored in the image and signal processing communities. For example, Melloni et al. develop a pipeline that identifies and aligns near-duplicate videos [40], and Pinheiro et al. introduce a fingerprinting method to identify correlations among near-duplicate videos [44]. However, unlike VSS, these techniques assume that sets of near-duplicate videos are known a priori and they do not exploit redundancies to improve compression or read/write performance. Finally, the multiview extension to HEVC (MV-HEVC; similar extensions exist for other codecs) attempts to exploit spatial similarity in similar videos to improve compression performance [17]. These extensions are complementary to VSS, which could incorporate them as an additional compression codec for jointly-compressed video.

Finally, the database community has a long history of exploiting data independence to improve performance, a key technique used by VSS. For example, VSS transparently substitutes Zstandard [13] for a video codec. Other orthogonal optimizations could further improve performance (e.g., Vignette [39] or homomorphic operators [19]).

8 CONCLUSION

We presented VSS, a video storage system designed to improve the performance of video-oriented applications and data management systems. VSS decouples high-level operations such as machine learning algorithms from the low-level plumbing to read and write data in a suitable format. Users leverage VSS by reading and writing video data in their chosen format, and VSS transparently identifies the most efficient method to retrieve that video data.

As future work, we will extend VSS's joint compression optimization to support more intelligent techniques for merging overlapping pixels. For example, rather than simply overlaying pixels from one camera onto another, VSS might intelligently detect occlusions and keep both pixels in these areas. This is important for cases where video must be maintained in its (near-)original form (e.g., for legal reasons). Our experiments showed that, relative to local storage and other video storage systems, VSS offers more flexible formats, reduces read time by up to 54%, and decreases the cost of persisting video by up to 45%.

REFERENCES

[1] Amazon. 2020. Serverless Image Handler. https://aws.amazon.com/solutions/implementations/serverless-image-handler.

[2] Ganesh Ananthanarayanan, Victor Bahl, Landon P. Cox, Alex Crown, Shadi Nogbahi, and Yuanchao Shu. 2019. Video Analytics - Killer App for Edge Computing. In *MobiSys*. 695–696.

[3] Michael R. Anderson, Michael J. Cafarella, German Ros, and Thomas F. Wenisch. 2019. Physical Representation-Based Predicate Optimization for a Visual Analytics Database. In *ICDE*. IEEE, 1466–1477.

[4] Favyen Bastani, Songtao He, Arjun Balasingam, Karthik Gopalakrishnan, Mohammad Alizadeh, Hari Balakrishnan, Michael J. Cafarella, Tim Kraska, and Sam Madden. 2020. MIRIS: Fast Object Track Queries in Video. In *SIGMOD*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). 1907–1921.

[5] Favyen Bastani, Oscar R. Moll, and Samuel Madden. 2020. Vaas: Video Analytics At Scale. *VLDB* 13, 12 (2020), 2877–2880.

[6] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. 2010. Finding a Needle in Haystack: Facebook's Photo Storage. In *OSDI*. 47–60.

[7] Fabrice Bellard. 2018. FFmpeg. https://ffmpeg.org.

[8] Daniel S. Berger, Nathan Beckmann, and Mor Harchol-Balter. 2018. Practical Bounds on Optimal Caching with Variable Object Sizes. *POMACS* 2, 2 (2018), 32:1–32:38.

[9] Cloudview. 2018. Visual IoT: Where the IoT Cloud and Big Data Come Together. http://www.cloudview.co/downloads/10. (2018).

[10] Victor H. Costa, Pedro A. Amado Assunção, and Paulo J. Cordeiro. 2018. A pixel-based complexity model to estimate energy consumption in video decoders. In *ICCE*. 1–5.

[11] Maureen Daum, Brandon Haynes, Dong He, Amrita Mazumdar, Magdalena Balazinska, and Alvin Cheung. 2020. TASM: A Tile-Based Storage Manager for Video Analytics. arXiv:2006.02958

[12] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS*. 337–340.

[13] Facebook. [n. d.]. Zstandard real-time compression algorithm. https://facebook.github.io/zstd.

[14] David A. Forsyth and Jean Ponce. 2012. *Computer Vision - A Modern Approach, Second Edition*. Pitman.

[15] Vishakha Gupta-Cledat, Luis Remis, and Christina R. Strong. 2017. Addressing the Dark Side of Vision Research: Storage. In *HotStorage*.

[16] Alon Y. Halevy. 2001. Answering queries using views: A survey. *VLDB* 10, 4 (2001), 270–294.

[17] Miska M. Hannuksela, Ye Yan, Xuehui Huang, and Houqiang Li. 2015. Overview of the multiview high efficiency video coding (MV-HEVC) standard. In *ICIP*. 2154–2158.

[18] Brandon Haynes, Maureen Daum, Amrita Mazumdar, Magdalena Balazinska, Alvin Cheung, and Luis Ceze. 2020. VisualWorldDB: A DBMS for the Visual World. In *CIDR*.

[19] Brandon Haynes, Amrita Mazumdar, Armin Alaghi, Magdalena Balazinska, Luis Ceze, and Alvin Cheung. 2018. LightDB: A DBMS for Virtual Reality Video. *PVLDB* 11, 10 (2018), 1192–1205.

[20] Brandon Haynes, Amrita Mazumdar, Magdalena Balazinska, Luis Ceze, and Alvin Cheung. 2019. Visual Road: A Video Data Management Benchmark. In *SIGMOD*. 972–987.

[21] Stephan Heinrich and Lucid Motors. 2017. Flash memory in the emerging age of autonomy. *Flash Memory Summit* (2017).

[22] Alain Horé and Djemel Ziou. 2010. Image Quality Metrics: PSNR vs. SSIM. In *ICPR*. 2366–2369.

[23] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodík, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. 2018. Focus: Querying Large Video Datasets with Low Latency and Low Cost. In *OSDI*. 269–286.

[24] Chien-Chun Hung, Ganesh Ananthanarayanan, Peter Bodík, Leana Golubchik, Minlan Yu, Paramvir Bahl, and Matthai Philipose. 2018. VideoEdge: Processing Camera Streams using Hierarchical Clusters. In *SEC*. 115–131.

[25] Shelley S Hyland. 2018. Body-worn cameras in law enforcement agencies, 2016. *Bureau of Justice Statistics Publication No. NCJ251775* (2018).

[26] Samvit Jain, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, and Joseph Gonzalez. 2019. Scaling Video Analytics Systems to Large Camera Deployments. In *HotMobile*. 9–14.

[27] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodík, Siddhartha Sen, and Ion Stoica. 2018. Chameleon: scalable adaptation of video analytics. In *SIGCOMM*. 253–266.

[28] Junchen Jiang, Yuhao Zhou, Ganesh Ananthanarayanan, Yuanchao Shu, and Andrew A. Chien. 2019. Networked Cameras Are the New Big Data Clusters *(HotEdgeVideo'19)*. 1–7.

[29] Daniel Kang, Peter Bailis, and Matei Zaharia. 2019. BlazeIt: Optimizing Declarative Aggregation and Limit Queries for Neural Network-Based Video Analytics. *PVLDB* 13, 4 (2019), 533–546.

[30] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2017. NoScope: Optimizing Deep CNN-Based Queries over Video Streams at Scale. *PVLDB* 10, 11 (2017), 1586–1597.

[31] Daniel Kang, Ankit Mathur, Teja Veeramacheneni, Peter Bailis, and Matei Zaharia. 2020. Jointly Optimizing Preprocessing and Inference for DNN-based Visual Analytics.

[32] Sooyong Kang, Sungwoo Hong, and Youjip Won. 2009. Storage technique for real-time streaming of layered video. *Multimedia Systems* 15, 2 (2009), 63–81.

[33] Andrea Lottarini, Alex Ramírez, Joel Coburn, Martha A. Kim, Parthasarathy Ranganathan, Daniel Stodolsky, and Mark Wachsler. 2018. vbench: Benchmarking Video Transcoding in the Cloud. In *ASPLOS*. 797–809.

[34] David G. Lowe. 1999. Object Recognition from Local Scale-Invariant Features. In *ICCV*. 1150–1157.

[35] David G. Lowe. 2004. Distinctive Image Features from Scale-Invariant Keypoints. *IJCV* 60, 2 (2004), 91–110.