Data Models, Query Execution, and Storage for Traditional & Immersive Video Data Management

Brandon Haynes

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2020

Reading Committee:

Magdalena Balazinska, Chair

Alvin Cheung, Chair

Luis Ceze

Program Authorized to Offer Degree:
Paul G. Allen School of Computer Science & Engineering

University of Washington

Abstract

Data Models, Query Execution, and Storage for
Traditional & Immersive Video Data Management

Brandon Haynes

Co-Chairs of the Supervisory Committee:
Professor Magdalena Balazinska
Paul G. Allen School of Computer Science & Engineering, University of Washington

Assistant Professor Alvin Cheung
Department of Electrical Engineering and Computer Sciences, University of California, Berkeley

The proliferation of cameras deployed throughout our world is enabling and accelerating exciting new applications such as virtual and augmented reality (VR and AR), autonomous driving, drone analytics, and smart infrastructure. However, these cameras collectively produce staggering quantities of video data. VR spherical (360°) video is up to 20× larger in size than its 2D counterparts. Closed-circuit television camera networks, often consisting of tens of thousands of cameras, generate petabytes of video data per day. A single autonomous vehicle can generate tens of terabytes of video data per hour.

Due to these massive data sizes and the complexity involved with reasoning about large numbers of cameras, developing applications that use real world video data remains challenging. Developers must be cognizant of the low-level storage intricacies of video formats and compression. They need expertise in device-specific programming (e.g., GPUs), and, to maximize performance, they must be able to balance execution across heterogeneous, possibly distributed hardware.

In this thesis, we describe several video data management systems designed to simplify application development, optimize execution, evaluate performance, and move forward the state of the art in video data management. The first system, LightDB, presents a simple, declarative interface for VR and AR video application development. It implements powerful query optimization techniques, an efficient storage manager, and a suite of novel physical optimizations. To further improve the performance of video applications, we next introduce a new video file system (VFS), which can serve as a storage manager for video data management systems (such as LightDB and others) or can be used as a standalone system. It is designed to decouple video application design from a video's underlying physical layout and compressed format. Finally, analogous to standardized benchmarks for other areas of data management research, we develop a new benchmark—Visual Road—aimed specifically at evaluating the performance and scalability of video-oriented data management systems and frameworks.

By exposing declarative interfaces, LightDB and VFS automatically produce efficient execution strategies that include leveraging heterogeneous hardware, operating directly on the compressed representation of video data, and improving video storage performance. Visual Road reproducibly and objectively measures how well a video system or framework executes a battery of microbenchmarks and real-world video-oriented workloads. Collectively through these systems, we show that the application of fundamental data management principles to this space vastly improves runtime performance (by up to 500× increased throughput), enhances storage performance (up to 45% decrease in file sizes), and greatly decreases application development complexity (decreasing lines of code by up to 90%).

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

Though my name is on this thesis, its contents would not have been possible without the help of many others. I am principally immensely grateful to my amazing advisors and mentors Magda Balazinska and Alvin Cheung. Both are true visionaries, possess unbounded enthusiasm, and carry limitless patience to tolerate my many missteps. None of this would have been possible without them, and I have learned so much from both. Equally amazing are my informal advisors (especially Dan Suciu and Bill Howe) and committee members (Luis Ceze), whose mentorship, perspective, insight, and feedback has been invaluable through the years.

I am extremely fortunate to have been part of an amazing database group during my time at the University of Washington. Sharing this chapter of my life with these amazing individuals has been life-altering. I would like to especially acknowledge my frequent collaborators—Maureen Daum, Amrita Mazumdar, and Dong He—for their fantastic support, feedback, ideas, and insight. I have benefited much from each of them. Equally important are the others who have tolerated my myriad inanities with grace and made this experience so fulfilling during my time here: Cong Yan, Jennifer Ortiz, Laurel Orr, Ryan Maas, Walter Cai, Guna Prasaad, Maaz Ahmad, Shrainik Jain, Jingjing Wang, Remy Wang, Moe Kayali, Shumo Chu, Srini Iyer, Tomer Kaftan, Babak Salimi, Batya Kenig, Max Schleich, and many others.

Other individuals have been instrumental in this journey, and I close by acknowledging some of them. Thanks to the initiate members of the arcane database underground reading group, through whom I mainlined the secret truths of the universe (and the chase). You know who you are. To Stacey, for her infinite patience, unlimited espresso, and for accompanying me on this journey into madness. To Jessie, for scotch, enduring my papers, and to summers gone by. To my cats, current and departed, for only occasionally jumping on my keyboard. Finally, to those I have inadvertently omitted, and to those lost to memory. So it goes.

DEDICATION

If you are lucky enough to have lived
in Paris as a young man, then wherever you
go for the rest of your life, it stays with
you, for Paris is a moveable feast.

Ernest Hemingway
to a friend, 1950

To Stacey, Mom, and Dad.

Chapter 1

INTRODUCTION

Video data management has recently re-emerged as an active research area due to advances in machine learning and graphics hardware, as well as the emergence of new video-oriented application domains such as virtual reality (VR) [47, 55, 153, 166, 100], augmented reality (AR) [65, 103], drone analytics [13, 27, 53, 154, 156, 157], autonomous driving [75, 146, 158, 172], and large-scale traffic and surveillance analytics [98, 169, 170, 172]. These new application domains differ from previous-generation video-oriented applications such as content-based search and adaptive streaming both in volume of video data being processed, the need to jointly reason about groups of videos (e.g., every drone camera in a swarm), and increased emphasis on each camera's physical location and orientation in the physical world.

Instances of these applications, which we call *visual world applications*, are abundant, and are typified by the following examples:

Example 1.1 (**Predictive spherical panoramic video tiling**)**.** Many streaming video services (e.g., YouTube VR [166]) stream extremely large, high-resolution 360° panoramic videos to client devices. This approach is suboptimal, because at any instant only a small portion of the panorama is displayed on a VR headset, which generally have a constrained field of view. Recent work [48, 56, 67, 106] has demonstrated substantial savings (up to 75%) in data transfer by degrading the quality of the "unimportant" areas of a 360° video. This example application is increasingly common today and it exemplifies the need to easily reason about, query, and process 360° panoramic video data.

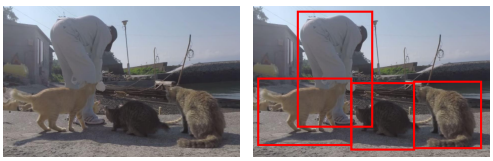


Figure 1.1: An augmented reality application that highlights detected objects (people and cats) on top of a live video stream. Image by Mewpro, CC-BY 3.0.

Example 1.2 (**Augmented reality object detection**)**.** An augmented reality application that has seen heightened interest in the computer vision [129, 95] and mobile sensing [26, 128] communities involves ingesting a live video stream from a worn camera or mobile device, automatically detecting salient objects within the viewer's field of view, and highlighting them in real-time with a bounding rectangle. Figure 1.1 shows the input and output of this process for a single video frame. Developers who write augmented reality applications must tediously manage details such as video compression and resolution, which motivates a need to abstract these physical details. These details can increase the size of programs by more than 10× (in terms of lines of code; see Section 3.5.1).

Example 1.3 (**Surveillance camera networks & autonomous driving**)**.** Large networks of city-wide surveillance cameras are an increasingly common occurrence. They require various types of processing large numbers of videos in various locations and orientations. An example application in this area might involve tracking vehicle sightings throughout a city (e.g., identifying vehicles across all cameras and assembling a montage of each video segment containing a given license plate) for offline viewing. At the same time, autonomous driving applications consume video in various formats (e.g., LiDAR, RADAR, stereoscopic cameras) and spatial configurations (e.g., monoscopic, stereoscopic), reason about its position relative to nearby objects, and may be augmented by information provided by other agents (e.g., nearby cars, surveillance cameras, or other intelligent transportation infrastructure) [172]. In both cases, joining together and reasoning about multiple videos is complex and difficult

in existing video processing systems, both because of varying physical formats (e.g., frame layouts include RGB, RGBA, NV12, YUV420, YUV422, and many others) and a lack of support for spatiotemporal position and orientation. This motivates a need for systems that can more easily reason about many videos captured from proximate locations in a format-agnostic manner.

These new application domains have led to many new systems that efficiently process and manage video data. This has led to new video-oriented data management systems, platforms, and frameworks (collectively "VDBMSs") that enable developers to more easily implement and execute applications such as those described above. These new systems target areas such as 2D analytics [98, 124, 6, 89, 83, 169] and machine-learning [73, 86, 85].

However, current VDBMSs—including all the systems referenced above—process individual video data streams in isolation, leaving the developer responsible for combining, overlaying, and intermixing videos per her requirements. They do not expose high-level support for, or reasoning about, a video source's "real world" characteristics such as dynamic spatiotemporal position or visual overlap between cameras. They additionally require tedious management of low-level details such as resolution and frame rate. Finally, with the exception of recent VR and AR systems (e.g. [65]), current systems do not expose support for reasoning about a viewer or video's orientation or perspective. However, even these VR and AR systems allow reasoning only about the angular orientation of a "current viewer" on a single client device or headset, and lack support for spatial reasoning, multiple video inputs, and multiple viewers.

Despite this lack of support, an important class of emerging video-oriented applications (including Examples 1.1 to 1.3) rely on *exactly* these "real world" concepts such as spatiotemporal position, orientation, and field of view. These *visual world applications* (VWAs) require reasoning about or operation over one or more of these additional concepts. They are thus distinguished from previous-generation video-oriented applications such as adaptive content delivery [80, 7], content retrieval [41, 1], and those that perform modifications to groups of location-agnostic, largely independent video streams [124, 57].

While implementing VWAs on many current VDBMSs is possible, doing so requires overcoming a number of conceptual and technical challenges, and leads to brittle implementations that intermix application logic with the plumbing required to address these challenges. This *VWA impedance mismatch* is typified by the following disconnects and challenges:

Definition 1.1 (**VWA impedance mismatch**)**.** Expressing and implementing VWAs using existing VDBMSs is impeded by the following factors:

Lack of uniform coordinate system. Unlike previous-generation video applications that process the input iteratively by independent video frames, VWAs emphasize and need to reason about viewer, camera, or object position, whether relative or absolute.

No support for orientation and overlapping field of view. VWAs often require joint reasoning about the positions of sets of cameras or viewers. For example, a VWA might apply object recognition to identify the *same* object in multiple, overlapping cameras, select all cameras that can see a particular point in three-dimensional space, or decrease the quality of video outside of a viewer's current field of view.

Physical data dependence. VWAs often require joining together multiple videos taken from different perspectives (e.g., multiple physically-proximate traffic cameras). However, developers who use current VDBMSs have to consider details about video in its physical format, and must manually account for physical differences *between* videos in terms of resolution, frame rate, pixel formats (e.g., NV12 [143]), and video codec idiosyncrasies (e.g., [110]). For VR and AR video, developers must additionally deal with issues such as spherical projections (e.g., [21, 133]), nonuniform sampling [19], and angular periodicity.

Opaque, coarse-grained storage. File system support for interacting with video data is limited to coarse-grained reads and writes of separately-stored video files. These opaquely compressed files have no support for co-locating, indexing, or caching spatially-similar video data, which is common in VWAs.

Compounding this, the performance and scalability of current VDBMS are evaluated in an ad hoc manner. Such evaluation requires large volumes of globally-consistent, realistic, high-resolution video datasets in a variety of formats, resolutions, and with accurate ground truth. Comparing scalability and performance *between* VDBMSs requires a common set of queries and operations found in VWAs, but objective measures for this evaluation are missing.

Finally, in addition to the impedance mismatch that hinders *implementing* VWAs, *optimizing* execution in these systems is also a substantial engineering challenge. While all video applications tend to be data-intensive and latency-sensitive, VWAs are often particularly so. The need to reason about position and orientation often requires joining together multiple video streams (e.g., two traffic cameras that can see the same intersection), resulting in higher complexity (and opportunities for optimization) compared to applications that process video streams independently and potentially in parallel. Additionally, operations in real or virtual space often requires computation over video metadata such as depth, increasing the cost of such operations.

1.1 Summary of Research Contributions

We critically need new and innovative video-oriented data models, data management systems, and query execution and optimization techniques to address the impedance mismatch found in VDBMSs when expressing, executing, and optimizing VWAs. With this in mind, this thesis presents new innovations in expressing, optimizing, and executing these applications and in evaluating the performance of systems they run on.

Specifically, our core contribution is a set of three systems—LightDB, VFS, and Visual Road—each designed to explore this space from a different perspective and level of the system stack. As illustrated in Figure 1.2, these systems, summarized below and detailed in ensuing chapters, build upon each other and collectively form a cohesive software stack for applications in this space. Collectively, this thesis contributes novel and application-driven approaches to *expressing*, *executing*, and *evaluating* both new and future VWAs, branching disconnects between the requirements of VDBMSs and the operations they support.

We next describe each contribution in further detail.

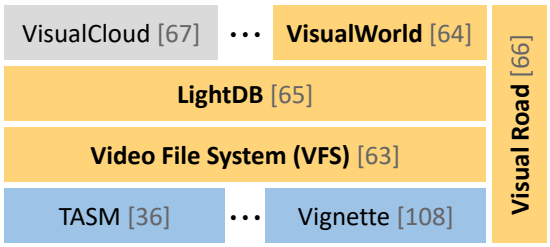


Figure 1.2: Systems developed in the context of this thesis (,), along with related contributions led by collaborators ( ), form a cohesive software platform. This monograph focuses on the systems highlighted in yellow ( ).

1.1.1 LightDB: A Database Management System for Virtual & Augmented Reality Video

As described previously, advances in computing, network hardware, and display technologies have generated increased interest in immersive VR and AR video applications. These applications have rapidly become mainstream and widely deployed on mobile and other consumer devices. As a result, a number of specialized systems have been introduced for preparing and serving VR and AR video data (e.g., [153, 47, 166, 55, 100, 103]), with the goal of aiding application development. However, developers who use current VR and AR systems still must overcome the challenges defined in the VWA impedance mismatch.

To address these challenges, we introduce the *LightDB* system (Chapter 3), which is a VDBMS designed to handle the storage, retrieval, and processing of both archived and live VR and AR video. This system was first demonstrated at SIGMOD'17 [67] and later appeared in PVLDB'18 [65].

A key contribution of LightDB is its unified data model, which contains a logical construct called a *temporal light field* (TLF). A TLF captures the degrees of freedom available to a human viewer in virtual space and serves as an abstraction over the various physical forms of VR and AR video that have been proposed.

In addition to a logically-unified data model, LightDB includes a logical algebra, data storage, query optimization, and execution components, along with a language that allows developers to easily write declarative queries in the LightDB algebra. We show implementation of real-world workloads using the LightDB language, and demonstrate that their expression requires far fewer lines of code than existing systems.

We have implemented LightDB as a prototype system that includes the TLF data model, algebra, and language. Our LightDB prototype comes with a storage manager, indexes, physical algebra, query language, and simple rule-based query optimizer. During execution, LightDB automatically selects an execution strategy that takes advantage of a number of optimizations for VR and AR VWAs. These execution strategies exploit heterogeneous hardware and device parallelism. Additionally, where possible, LightDB performs operations directly on compressed video and thereby avoids extremely expensive encoding and decoding costs.

We have used LightDB to implement a variety of real-world workloads, with significant improvement in application performance as compared to those implemented using existing VDBMSs and video processing frameworks. In particular, our microbenchmarks show orders of magnitude improvement (up to 500×) in terms of throughput for fine-grained operations, and up to 4× throughput performance improvement for end-to-end VR and AR video applications.

1.1.2 VFS: A File System for Video Analytics

To mitigate the video data deluge described earlier in this chapter, many VDBMSs have emerged to ingest, transform, and reason about video data [98, 65, 83, 124, 73, 86, 169]. These systems, however, treat video data as large, opaque blobs and stores them on persistent storage as independent files. As a result, storage-related operations on video data are inflexible and limited to reads, writes, and coarse-grained seeks. These limited storage capabilities create the following challenges for application developers:

- Developers are forced to tightly intermingle data plumbing with application logic and must manually handle the details associated with physically-persisted video data (e.g., (de)compression, resolution resampling, framerate alignment).
- The close coupling of application logic with physical video data storage makes it difficult to deploy optimizations and other techniques (including systems such as TASM [36] that are introduced after an application has been implemented or deployed) to improve application performance.
- As described previously, many recent VWAs collect large amounts of video data with overlapping fields of view and physically proximate locations. However, reducing the redundancies that occur among these sets of physically-proximate or otherwise similar video streams is neglected in all modern video-oriented systems.

To address these issues, we introduce a new *video file system (VFS)*, which is designed to decouple VWA design from video data's physical layout and compression optimizations. This decoupling allows application and system developers to focus on their relevant functionality, while VFS handles the low-level details associated with video data persistence and retrieval. In this way VFS takes responsibility for addressing the challenges listed above, freeing the developer to focus on relevant application logic.

Analogous to relational database management systems, developers using VFS treat each video as a *logical video* and let VFS determine the best way to perform operations over one or more of the *physical videos* that it maintains on disk. To enable this, VFS exposes a simple declarative interface where users read and write video data through file system operations, command line, or by directly utilizing its API. Users initially write video data in any format, encoding, and resolution and VFS manages the underlying compression, serialization, and physical layout on disk. For subsequent reads, VFS automatically identifies and leverages the most efficient methods to retrieve and return the requested data.

To transparently improve performance, VFS deploys a number of optimizations and caching mechanisms to improve read and write performance. For example, VFS indexes video fragments and the resulting structure allow VFS to read only the minimal set of subsequences necessary to satisfy a request for portions of a video.

Our results indicate that utilizing VFS can improve read performance by up to 54% (by materializing results and using them to answer future queries), reduce storage costs by up to 45% (by reducing redundancies in pairs of overlapping videos), and enable developers to focus on application logic rather than video storage and retrieval.

1.1.3 Visual Road: A Video Data Management Benchmark

When existing VDBMSs quantify their performance, they do so by reporting efficiency under various ad hoc workloads both in terms of the input videos selected and the executed queries. These ad hoc workloads, however, preclude comprehensive and easily reproducible system comparisons. This leads to a substantial challenge: there is no clear way to reliably and objectively benchmark performance among the many recently-proposed systems. This deficiency is due to a lack of a robust, sufficiently-complex video dataset, an architecture-agnostic set of queries that may be executed on current and future VDBMSs, and the difficulties in determining whether a VDBMS produces a correct answer to a query, which requires accurate ground truth that is laborious to generate.

To remedy these issues, and analogous to standardized benchmarks for other areas of data management research, we introduce a new benchmark called *Visual Road*. Visual Road is a benchmark aimed at VDBMSs and is designed to objectively evaluate the relative performance and scalability of modern video processing systems. This work appeared at SIGMOD'19 [66].

Visual Road comes with a dataset generator and a set of evaluation queries (expressed using the declarative LightDB query algebra formally defined in Section 3.2). These queries are divided into "microbenchmark" operations that test isolated features found in current VDBMSs, along with larger "composite" queries that measure a VDBMS's ability to execute typical end-to-end applications drawn from the recent literature.

Visual Road automatically generates an unlimited amount of realistic, synthetic VWA video data. To do so, Visual Road leverages a modern simulation, visualization, and gaming engine [42] to deterministically generate realistic videos within a simulated metropolitan world. In addition to allowing unlimited size and resolution for the resulting video datasets, this approach additionally allows for the *automatic* calculation of precise ground truth and other metadata about generated videos, without the need for manual annotation.

Through the composition of small operations available in all current VDBMSs (e.g., image cropping), the Visual Road query suite is designed to be implementable across a wide variety of VDBMS architectures. Further, and in the same way that relational database systems target subsets of benchmarks (e.g., a specific TPC query), Visual Road is flexible such that a user may either select specific applicable queries or groups of queries appropriate for their systems in addition to executing the entire benchmark to demonstrate broad functionality.

We show the utility of the benchmark by evaluating three recent VDBMSs both in capabilities and performance and demonstrate that our approach enables performance comparisons between VDBMSs with results that are more reliable and scalable than current approaches (e.g., small ad hoc datasets or randomly generated videos).

1.2 Thesis Organization

The remainder of this thesis is organized as follows. In Chapter 2 we describe the minimal background in video capture and storage necessary to understand the details in this monograph. In Chapter 3 we introduce LightDB, a database management system designed for VR and AR video applications. Chapter 4 describes VFS, a specialized file system designed for video analytics. Chapter 5 describes Visual Road, a benchmark for evaluating the performance and scalability of VDBMSs. We then describe related work in Chapter 6 and conclude in Chapter 7.

Chapter 2

OVERVIEW OF VIDEO CAPTURE & STORAGE

In this chapter we briefly summarize the concepts and terminology relating to video capture, compression, and format. This background is not intended to be comprehensive, and we refer a reader to a more robust treatment of these topics (e.g., [133, 145]) for details beyond the scope of this monograph. We begin in Section 2.1 by briefly describing how cameras capture and store visual information. Section 2.2 introduces techniques relating to video encoding and streaming, and Section 2.3 discusses the terms and concepts necessary for managing virtual reality video.

2.1 Photographic & Video Camera Physical Structure

Cameras record visual information in the form of photographs or videos. To capture this visual information, most digital cameras contain one or more lenses that focus light onto an image sensor. Each image sensor can be conceptualized as an $n \times m$ array with elements that correspond to pixels in a photograph or video.

All cameras have a *field of view* that determine the angular extent visible on the resulting photograph or video. By way of example, a 35mm camera typically has a horizontal field of view of 39.6°. Spherical panoramic videos, described further in Section 2.3, are often referred to as 360° videos because they have a maximal field of view and capture visual information from every direction.

The mapping between a light ray incident to a camera at a particular angle (θ, ϕ) (i.e., the azimuth and altitude) and the corresponding image sensor array element (i, j) is a function of the lenses and other physical properties of the camera. For two-dimensional photography, this is a straightforward linear *forward projection* and illustrated in Figure 2.1(a). For cameras

(a) **Photographic camera.** A two-dimensional photographic camera with a field of view defined by its lens and other physical characteristics. A forward projection is used to map light rays onto a plane.

(b) **Spherical panoramic camera.** A spherical panoramic camera with dual fish-eye lenses used to capture visual information from every angle. An equirectangular projection is used to map light rays in every direction onto a plane.

(c) **Light field camera.** An $n \times m$ bank of light field cameras (see Section 2.3.2) captures visual information incident to a plane. A light field projection maps each light ray onto a plane.

Figure 2.1: High-level illustration of a photographic, spherical, and light field cameras capturing light from their respective field of view and projecting it onto an image plane.

with a maximal field of view such as spherical panoramas and light fields (see Section 2.3), more sophisticated projections are used. Figure 2.1(b) shows a spherical panoramic camera with a dual fish-eye lenses using an equirectangular projection function to map spherical visual information onto a plane. Finally, as illustrated in Figure 2.1(c), a light field camera uses an array of ordinary cameras and a complex projection to map the substantially larger amount of visual information onto a plane.

2.2 Video Encoding & Streaming

A video is logically composed of a sequence of *video frames* that are periodic temporal samples of visual data. Each frame is an independent image sampled at a point in time. Common *frame rates* generally fall between 25 and 150 frames per second. Each frame is a two-dimensional array with extents given by its *resolution* $R = (R_x, R_y)$. Typical resolutions

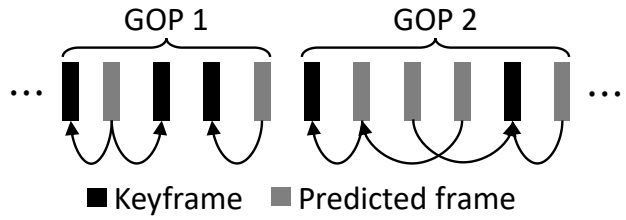


Figure 2.2: Illustration showing keyframes (a.k.a. intra-coded frame or I-frame), which are compressed in isolation, and predicted frames that must be decompressed in conjunction with their dependencies (shown by arrows between frames). Two groups of pictures (GOPs)—a sequence of frames with no external dependencies–are also labeled.

are 1K (960×540), 2K (1920×1080), and 4K (3840×2160), and a resolution is typically fixed for a given video. Each element of a frame is a pixel containing an encoded color (e.g., YUV420p) in a given *color space* (e.g., CMYK).

Naively serializing a video as described above would be cost-prohibitive, as uncompressed 4K video consumes approximately 7GB of space per hour of video. To reduce storage and transmission costs, videos are encoded and compressed using a *video codec* (e.g., H264 [160], HEVC [144]). Most modern video codecs compress each frame as a *keyframe* (a.k.a. intra-coded frame or I-frame) or a *predicted frame* (a.k.a. P- or B-frame). Keyframes are compressed in isolation and may thereafter be decoded independently without reference to any other frame. Predicted frames take advantage of redundant visual information between frames and maintain references to other frames to improve compression performance. As a consequence, each predicted frame must be decoded in conjunction with its dependent frames. These dependencies are illustrated in Figure 2.2.

Video codecs also use redundant information within each frame to improve compression by identifying regions with high similarity and storing each region only once. While codecs may search across the entire frame when looking for similar regions, it is often useful to restrict this search to *tiles* (equivalent to *slices* for the purposes of this discussion) within a frame. While this technique—called *motion constrained tile sets*—reduces compression

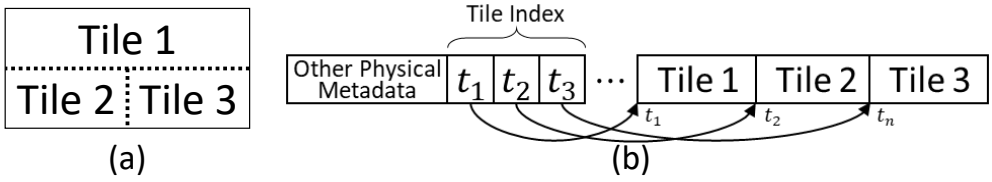


Figure 2.3: In (a), a frame is subdivided into three independently-decodable tiles. Subfigure (b) shows its corresponding physical representation, where an index points to the offset of each tile.

performance, it enables tiles to be (de)compressed in parallel and allows other tiles (e.g., of different qualities) to be substituted without affecting the rest of the frame. As shown in Figure 2.3, state-of-the-art video codecs (e.g., HEVC, VP9) associate a *tile index* with each frame that utilizes tiles. This index allows for rapid identification of the data region associated with each tile.

Many video codec implementations produce *groups of pictures (GOPs)* that are independently decodable as a group and begin with a synchronizing keyframe. This means that within a GOP, every predicted frame depends only on frames within that GOP. Two GOPs are shown in Figure 2.2. GOPs are an important part of *adaptive streaming* (e.g., DASH [80], HLS [7]), which varies the quality of each GOP delivered to a client based upon its current network conditions [137].

Finally, rather than streaming raw encoded video streams to clients, videos are typically "muxed" into files such as the MPEG-4 Part-14 (MP4) [79] or WebM/Matroska [147] media container formats. These "containers" standardize a flexible format for video and audio metadata, support aggregation of multiple streams into a single file structure (e.g., different audio languages or camera perspectives), and allow for data indexing.

An MP4 file may be associated with any number of *tracks* that contain independent video streams. For example, as we will describe in Section 3.4.4, our prototype LightDB system stores stereoscopic visual information for the left and right eyes in separate video streams. Additionally, if depth map information is available, LightDB stores it as a separate video stream.

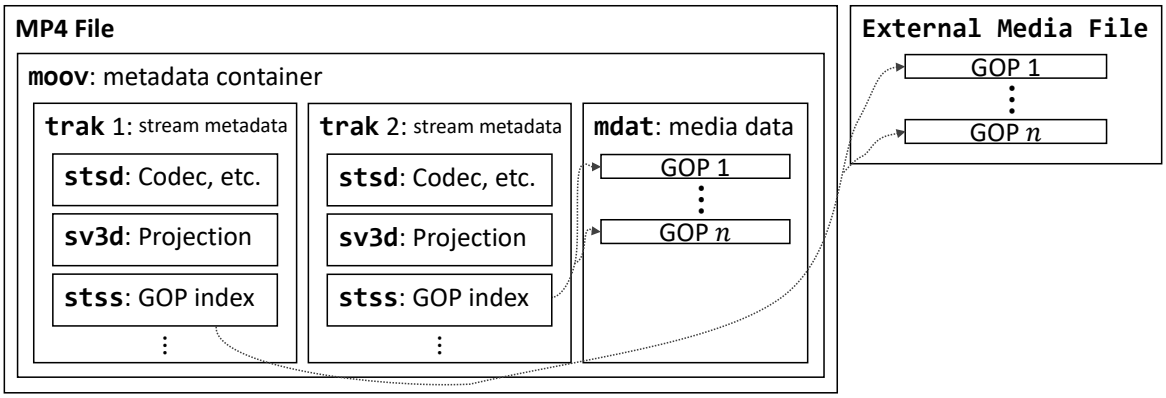


Figure 2.4: Abridged MP4 layout showing atoms relevant to this monograph. This MP4 file contains a `moov` atom holding media metadata. This includes two `trak` atoms, each containing metadata about a single media stream. A `stss` atom provides a GOP index over media data stored in a `mdat` atom or externally.

As illustrated in Figure 2.4, an MP4 file is composed of a forest of *atoms* (often called "boxes"). An atom is a self-contained data unit that contains information about media. Each atom is associated with a four-character identifier that indicates its type and a variable-length data region. Examples relevant to the discussion herein include:

- `mdat` atoms, which hold actual encoded media data.
- A `trak` atom, which holds metadata about a particular media stream (e.g., its codec) and a pointer to media data. Actual media data may be stored externally (`trak1` in Figure 2.4) or embedded inside a `mdat` atom (`trak2`).
- The `stss` atom contains an index of the GOPs in a video stream, and is used to efficiently look up the beginning of any GOP without needing to linearly search through the encoded video data.
- The `sv3d` atom is a custom atom defined in the Spherical Video V2 RFC [141] and is used to store metadata relevant to virtual and augmented reality video (see Section 3.4.4).

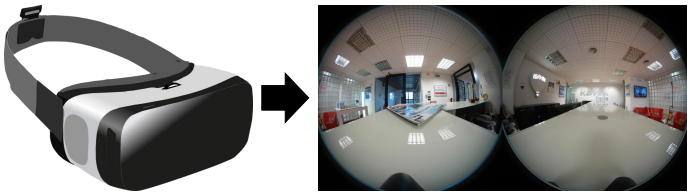


Figure 2.5: A VR head-mounted display (left) and an example of the stereoscopic 360° video image projected to the viewer (right).

2.3 Light Fields & Spherical Panoramas

In this section we discuss the details of the most popular forms of virtual reality video: *spherical panoramic* (a.k.a. 360°) and *light field* videos. In Chapter 3 we describe LightDB's support for both video formats, while in Chapter 5 we describe Visual Road's support for evaluating the performance of producing and operating on 360° workloads.

Ultimately both 360° and light field videos are encoded using the two-dimensional video techniques described in Section 2.2 to reduce the amount of storage required. The following two subsections introduce each type of video and highlight various techniques applied prior to encoding.

2.3.1 Spherical (360°) Panoramas

360° videos are a popular form of VR video that allow a viewer wearing a head-mounted display or using a mobile device to observe a scene from a fixed location at any angle. Figure 2.5 shows a typical mobile headset device and a corresponding 360° view. Because a user may rapidly adjust the direction of view, the critical variable for this format is the direction that a user is looking. *Spherical panoramic images* are a special case of a 360° video where the scene is captured at a single instant of time.

As shown in Figure 2.6, the common approach to generate 360° video is to use multiple input cameras and stitch the streams together using specialized software that approximates a spherical representation. Rather than attempting to compress each sphere as a three-

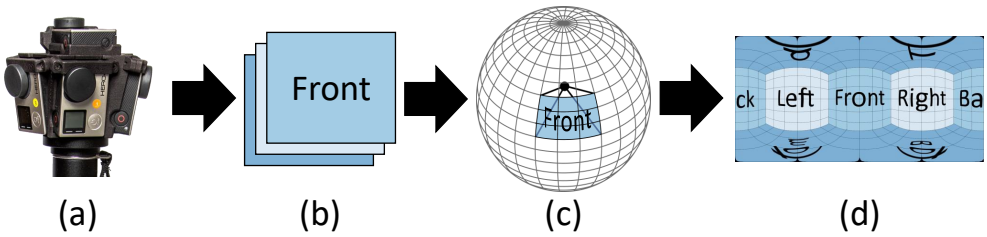


Figure 2.6: A typical 360° video ingest pipeline. In (a-b), an input camera rig generates overlapping 2D images in every direction. In (c-d) the 2D images are stitched into a spherical representation and then equirectangularly projected onto a 2D plane [67].

dimensional construct, the typical approach projects each sphere onto a two-dimensional plane using a specialized projection function and then compresses it using an ordinary 2D video codec. Common projections are equirectangular (ER) [133], cubic [133], or equiangular cubic [21].

360° images and videos may be monoscopic or stereoscopic. Stereoscopic videos (shown in Figure 2.5) encode visual data from two spatially-nearby points—the distance between a viewer's eyes. This encoding may be explicit, where two separate spheres are mapped onto two planes and delivered to viewers as separate video streams. Alternatively, if depth information is available (either from the capture devices or by applying an estimation algorithm), this may be encoded as a separate stream (i.e., a *depth map* [134]) and delivered to a viewer. The viewer uses the depth information to generate and render stereoscopic spherical images locally.

2.3.2 Light Fields

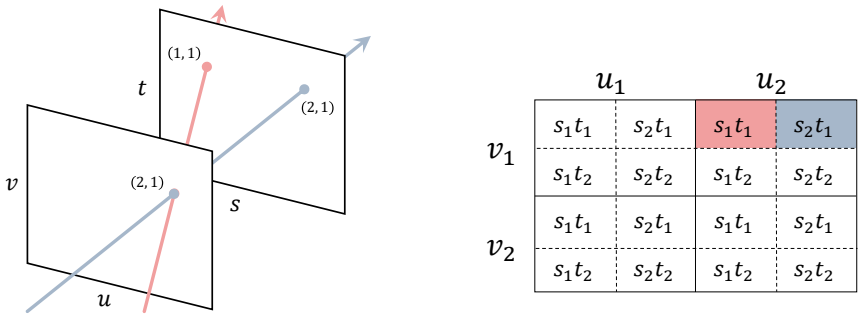
A 360° video enables a user to look in any direction, but the user must remain static. A *light field* enables a user to both look in any direction *and* move about in space. Obviously, this requires knowing the location and orientation of a viewer.

To enable such flexibility, a light field is a function that, for any point in a given volume and for any viewing direction, returns the color and amount of light flowing into a user's eye. Building a light field function requires a matrix of cameras to sample all of the light flowing through a volume in space. Figure 2.1(c) shows an example of this camera matrix.

One method of encoding light field data is to use a *light slab* [91], which encodes the color of each light ray by its intersection at points (i, j) and (k, l) with two planes uv and st. Multiple light slabs at various orientations are used to capture an entire volume. For example, six slabs could be used to capture the light in a cube. Figure 2.7(a), adapted from Levoy & Hanrahan [91], shows the uv and st planes of a single slab.

As with 360° videos, data from the uv and st planes can be projected onto a single two-dimensional plane and compressed using standard 2D video encoders. One common projection technique, used herein and shown in Figure 2.7(b), encodes each light ray color as an array of arrays [91]. For a given light slab S, the intersection (i, j) with the uv plane is used as a lookup in the outer array, and the intersection (k, l) with the st plane is used to look up the color in the nested array. Entry $S[i, j][k, l]$ gives this color.

Figure 2.7(b) shows a 2×2 sampling of a light field (typical light slabs have many more samples). The red and blue rays illustrated in Figure 2.7(a) are highlighted respectively at entries $S[u{=}2, v{=}1][s{=}1, t{=}1]$ and $S[u{=}2, v{=}1][s{=}1, t{=}2]$. When a viewer wishes to render a pixel for a user's position and orientation, a corresponding light ray is retrieved from this representation. To render rays that fall between samples, nearby rays are extracted and an interpolation function approximates the original light ray. When multiple light slabs are present, each is encoded into its own physical array and multiple slabs may be used during interpolation.



(a) Logical encoding (b) Physical encoding

Figure 2.7: Logical and physical encoding of a light slab with a 2×2 sampling of the uv and st planes. Two equivalent light rays are highlighted in red and blue.

Chapter 3

LIGHTDB: A DATABASE MANAGEMENT SYSTEM FOR VIRTUAL & AUGMENTED REALITY VIDEO

In this chapter we introduce the LightDB video database management system (VDBMS). LightDB's design targets the virtual reality (VR) and augmented reality (AR) domains—an important subset of the visual world applications (VWAs) described in Chapter 1. LightDB includes support for both spherical panoramic (360°) and light field video (see Section 2.3). We use LightDB to implement recent virtual and augmented reality applications and show that their performance (in terms of programmability and throughput) far exceeds that of other state of the art VDBMSs and video processing frameworks. The work presented in this chapter appeared in PVLDB'18 [65].

Over the last several years, advances in computing, network hardware, and display technologies have generated increased interest in immersive 3D VR video applications. AR applications, which intermix 3D video with the world around a viewer, have gained similar attention. Collectively, these VR and AR applications have become mainstream, widely deployed on mobile and other consumer devices, and represent an exciting and important subset of recent VWAs.

Managing VR and AR data at scale is an increasingly critical challenge. While, as we highlighted in Chapter 1, VWAs tend to be data-intensive and time-sensitive, VR and AR video applications are often particularly so. For example, recent VR light field cameras (introduced in Section 2.3.2), which sample every visible light ray occurring within some volume of space, can produce up to a *half terabyte per second* of video data [101, 109].

For spherical panoramic VR videos (a.k.a. 360° videos; see Section 2.3.1), encoding one stereoscopic frame of video can involve processing up to 18× more bytes than an ordinary 2D video [67].

AR video applications, on the other hand, often mix smaller amounts of synthetic video with the world around a user. Similar to VR, however, these applications have extremely demanding latency and throughput requirements since they must react to the real world in real time.

To address these challenges, various dedicated frameworks have been introduced for preparing and serving VR and AR video data (e.g., [153, 47, 166, 55, 100, 103]), with the goal of enabling developers to easily implement their applications. All current systems, however, suffer from the VWA impedance mismatch we described in Chapter 1 (see Definition 1.1): developers who use these systems must reason about 2D encoded video in a 3D VR or AR world; consider physical details about data in its compressed 2D format; and manually account for projections, angular periodicity, nonuniform sampling [19], and video codec idiosyncrasies (e.g., [110]). This leads to brittle implementations that intermix application logic with the plumbing required to address this impedance.

Further compounding this, current VR and AR systems are highly physically data dependent. For instance, we are aware of no other 360° system that is able to accept light field data, nor any light field system able to accept 360° video data. Incompatibilities even exist between 360° systems due to differing stereoscopic representations, video codecs, and incompatible spherical projections. Unfortunately, transforming data from one 2D format to another is prohibitively expensive, and this limits interoperation between systems that otherwise would be compatible.

To address the VWA impedance mismatch in this area, we design, implement, and evaluate LightDB, a new type of VDMS specialized for this new type of VR and AR video data. LightDB treats all types of VR and AR video in a logically unified manner. It introduces a unified data model containing a logical construct that we call a *temporal light field* (TLF). A TLF captures the degrees of freedom available to a human viewer in (potentially augmented)

virtual space and serves as an abstraction over the various physical forms of VR and AR video that have been proposed [21, 91, 133]. Modeling VR and AR videos as TLFs allows developers to express their video operations as *declarative queries* over TLFs, and decouples the intent of a query from the plumbing and manual optimizations that developers need to implement when using existing VR and AR systems along with 2D video processing frameworks such as FFmpeg [17] and OpenCV [118]. Declarative queries also offer the opportunity to introduce query optimization techniques that improve video workload performance, as we show in this chapter.

Under the TLF data model, LightDB is designed to handle the storage, retrieval, and processing of both archived and live VR and AR video. It additionally includes a novel algebra; data storage, query optimization, and execution components; a language that allows developers to easily write declarative queries for VR- and AR-oriented VWAs; and an implementation of each of these contributions.

LightDB builds on recent work in multimedia [14, 70, 96, 127] and multidimensional array processing [15, 22, 120, 119]. Its physical operators combine the state of the art in array-oriented systems (e.g., multidimensional array representation, tiling) with recently-introduced optimizations by the multimedia and graphics communities such as motion-constrained tile sets [110] and light field representations [91].

To allow developers to express queries in its algebra, LightDB exposes a declarative query language, *VRQL*, for developers to use. LightDB automatically selects an execution strategy for VRQL queries that takes advantage of a number of optimizations. We have used VRQL to implement a variety of real-world workloads, with significant improvement in resulting application performance as compared to those implemented using existing VR and AR systems.

In summary, we make the following contributions through the LightDB system:

- We introduce the temporal light field (TLF) data model, which unifies various physical forms of VR and AR video data under a single logical abstraction.

- We introduce a logical algebra and query language (*VRQL*) designed to operate over TLFs, and describe real-world workloads using VRQL.

- We describe the architecture of LightDB, a prototype system that implements the TLF data model and VRQL. LightDB comes with a no-overwrite storage manager, indexes, a physical algebra, and a simple rule-based query optimizer for optimizing VRQL queries.

- We evaluate LightDB against other video processing frameworks and array-oriented database management systems and demonstrate that LightDB can offer up to a 500× increase in frames per second (FPS) in our microbenchmarks, and up to 4× increase in FPS for real-world workloads.

The remainder of this chapter is organized as follows. We begin by introducing LightDB's data model (Section 3.1) and algebra (Section 3.2). We then describe the LightDB query language, VRQL, and use it to implement several recent VR and AR applications (Section 3.3). Finally, we describe the LightDB architecture in Section 3.4 and evaluate its performance in Section 3.5.

3.1 Data Model

Existing database management systems specialized in the processing of image and video data, including RasDaMan [15], SciDB [22], and Oracle Multimedia [119], model image and video data as multidimensional arrays. These arrays typically have three dimensions: x, y, and t. As detailed in Chapter 1, we find this model ill-suited for VWAs in general and VR and AR applications specifically. In particular, while standard spatiotemporal dimensions can identify a pixel in a single, isolated video stream, VR and AR applications do not reason only in those terms.

As highlighted by the VWA impedance mismatch (see Definition 1.1), for VR and AR applications that operate on video data, the fundamental concepts are: *the location of cameras* and *the direction in which they are facing*. These concepts, illustrated in Figure 3.1, are elegantly captured by the light field abstraction presented in Chapter 2 (see Section 2.3.2).

We therefore adopt light fields as our fundamental construct and will show how it is broadly useful for expressing and executing VR and AR VWAs. Because video data (and its corresponding light field) changes over time, we use *temporal light fields* (TLFs) [2] to represent all data, whether originally ingested as an ordinary 2D photographic video, light field, or 360° video. Collectively the use of and operation over a TLF forms the basis of the *temporal light field data model* described in this chapter.

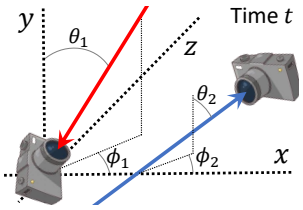
Concretely, we model video data as multidimensional objects with both rectangular and angular coordinates and six overall dimensions—three spatial (x, y, z), two angular (θ, ϕ), and one temporal (t). The spatiotemporal dimensions capture a camera or viewer's position over time while the angular dimensions capture orientation. This object is captured in the following definition:

Definition 3.1.1 (Temporal light field (TLF))**.** A TLF L is defined by a TLF function $L(x, y, z, t, \theta, \phi)$ that determines the color and luminance associated with light rays throughout a (possibly infinite) volume $V \subseteq \mathbb{R}^4 \times \mathcal{D}_\theta \times \mathcal{D}_\phi$. Application of L at points not in V produces the null token ω.

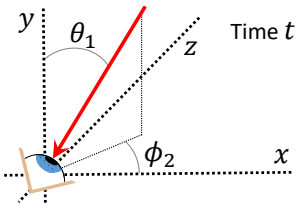
The domain $\mathcal{D}_{\text{TLF}}$ of a TLF is the product of the domains of each of its dimensions. For the spatiotemporal dimensions x, y, z, and t, this is the reals. The domain of angles θ and ϕ are respectively in the right-open range $[0, 2\pi)$ and $[0, \pi)$, which we denote $\mathcal{D}_\theta$ and $\mathcal{D}_\phi$.[1]

In the graphics community, the TLF function is called a *plenoptic function* [2]. Our TLF function formulation, which is equivalent to that proposed by Adelson and Bergen [2], is a function from position and orientation to a point in a user-specified color space C (e.g.,

[1]Ranging ϕ over $[0, 2\pi)$ would be ambiguous. For example, $(\frac{\pi}{2}, \pi)$ and $(\frac{3\pi}{2}, 0)$ identify the same point on a sphere.



(a) A temporal light field (TLF) L containing two cameras c_1 and c_2. At time t each camera respectively observes a red and blue light ray at angles (θ_1, ϕ_1) and (θ_2, ϕ_2).



(b) A viewer positioned at (x, y, z) in the TLF L defined in (a). At angles (θ_1, ϕ_1) she observes, at time t, the red light ray captured by camera c_1.

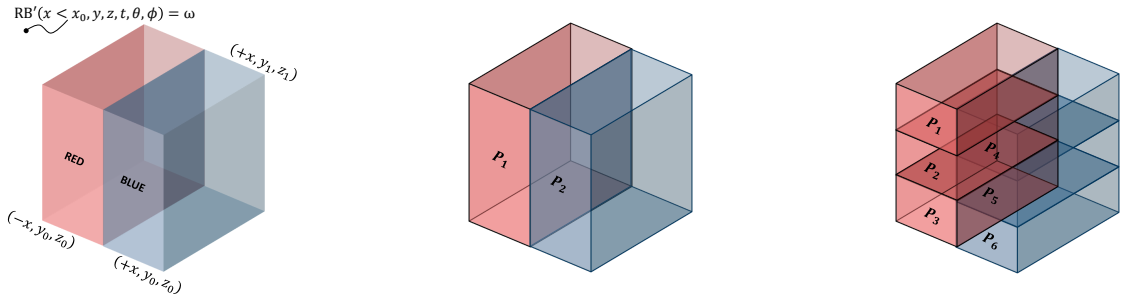
Figure 3.1: Illustration of a TLF L in three-dimensional space. In (a) at time t, rays from every angle in c_1 and c_2's field of view are captured. Two such rays—one red and one blue—are shown at angles (θ_1, ϕ_1) for c_1 and (θ_2, ϕ_2) for c_2. In (b), a subsequent viewer positioned at (x, y, z) observes the light rays previously captured by c_2.

RGB, CMYK). For example, consider a color space containing the colors RED and BLUE at a fixed intensity and a volume R defined by the points $(-x, y_0, z_0)$ and (x, y_1, z_1) (without constraining time or angles). The following TLF, illustrated in Figure 3.2(a), defines a field RB in R that is RED for all $x \leq 0$ and BLUE otherwise:

$$RB_R = \begin{cases} \text{RED} & \text{if } x \leq 0 \\ \text{BLUE} & \text{otherwise} \end{cases} \tag{3.1}$$

Data objects in LightDB take the form of temporal light fields. Every TLF L is also associated with metadata that includes a unique identifier and a bounding volume. We refer to them as $id(L)$ and $V(L)$.

TLFs may further be *partitioned* into pieces for parallel processing. For example, Figure 3.2(b) shows RB' as a possible partitioning of RB_R, where one partition contains the RED light rays and another contains the BLUE. Figure 3.2(c) shows a further subdivision of RB' into six equal-sized volumes with height $\frac{y_1 - y_0}{3}$.



(a) RB_R at time t bounded by $(-x, y_0, z_0)$ and $(+x, y_1, z_1)$.

(b) RB': RB_R partitioned on $x \leq 0$.

(c) RB'': A further partition of RB' along the y axis.

Figure 3.2: A temporal light field (TLF) and two possible partitionings.

To improve efficiency, we require that a TLF's volume and partitions be hyperrectangles (i.e., a six-dimensional analog of a rectangle). This partitioning information is also part of a TLF's metadata. Finally, TLF metadata includes a streaming flag to indicate whether its ending time monotonically increases (i.e., it is streaming) or is constant (e.g., a complete video serialized to disk).

3.2 Algebra

In this section we describe a query algebra over TLFs. This query algebra is designed to enable a variety of operations on different types of visual data to capture the logical specifications of those operations while hiding their physical complexities. For example, it abstracts the intricacies of physical video formats such as resolution, continuousness, interpolation, geometric projection, and sampling. To allow for easy composition and avoid the need for more complex transformations that produce or operate over TLF tuples, each operator accepts zero or more TLFs (along with other scalar parameters) and produces a single output TLF. Since TLFs are nullable and defined in a 6D space, developers need not be concerned with TLFs defined over different volumes.

The TLF algebra exposes nineteen logical operators for expressing queries. We classify them into three broad categories. First, we describe the data manipulation operators used to manipulate TLFs. Next, we present the input and output that are used to transform an internal TLF representation to and from an encoded representation (i.e., a MP4 file). Finally, we describe the data definition operators used to create, modify, and remove TLFs from persistent storage.

3.2.1 Data Manipulation

The data manipulation operators in the TLF query algebra include:

Selection. The SELECT operator derives a "smaller" TLF from its input. For example, Figure 3.2(a) illustrates selection over the TLF defined by Equation 3.1.

Similar to relational selection, this operator restricts the domain of a TLF L to some subset R (i.e., L_R). Unlike in relational algebra, TLF selection requires that the predicate over dimensions be restricted so that R be a well-defined hyperrectangle. We denote a selection of TLF L over a hyperrectangle R as:

$$\text{SELECT}(L, [x, x'], [y, y'], [z, z'], [t, t'], [\theta, \theta'], [\phi, \phi']) = L_R$$

Alternatively, a developer might wish to discretize a TLF at a regular interval (e.g., at 30 samples per second). The DISCRETIZE operator performs this operation by sampling a TLF over some interval Δd along a given dimension $d \in \{x, y, z, t, \theta, \phi\}$. It produces a new TLF with every point not on the interval set to null:

$$\text{DISCRETIZE}(L, \Delta d = \gamma) = L_{d \in \{i \cdot \gamma | i \in \mathbb{Z}\}}$$

As an example, a developer might reduce the physical size of a TLF by angular sampling at some resolution (e.g., 1920×1080) by invoking $\text{DISCRETIZE}(L, \Delta\theta = \frac{2\pi}{1920}, \Delta\phi = \frac{\pi}{1080})$.

Partitioning. The PARTITION operator "cuts" a TLF into equal-sized, non-overlapping blocks along a given dimension. For example, given an unpartitioned TLF L with duration of ten seconds. $\text{PARTITION}(L, \Delta t = 1)$ creates a TLF with ten one-second partitions. Figures 3.2(b)–(c) show further examples.

Inversely, the FLATTEN operator removes a TLF's partitions.

Merging. The UNION operator merges two or more TLFs into a single TLF. When inputs to UNION are non-overlapping, merging is unambiguous. However, an overlapping light ray may be present in one or more of the inputs. To resolve this ambiguity, when two inputs L_i and L_j are both non-null at point p, UNION applies a user-supplied merge function m to disambiguate as follows:

$$\text{UNION}(L_1, ..., L_n, m) = p \mapsto \begin{cases} L_i(p) \text{ if } \forall_{j \neq i} L_j(p) = \omega \\ m(L_1(p), ..., L_n(p)) \text{ otherwise} \end{cases}$$

Transformation. The MAP operator transforms a TLF into a new field defined within the same bounding volume as its input. Given a *transformation function* f, the MAP operator produces a new field with the color and luminance at each point replaced with the one given by the application of f. For example, Figure 3.5 shows use of the built-in SHARPEN filter.

The transformation function is parameterized by a point p and the source TLF (i.e., it is a function $f\colon (p, TLF) \to C$), where C is the color space of the TLF, and is defined as:

$$\text{MAP}(L, f) = p \mapsto f(p, L)$$

In practice, the above formulation requires that the entire TLF L be available during every invocation of f. This restriction makes parallelization difficult, since L may be expensive to transfer (e.g., CPU to GPU). However, many transformations only need a small region (i.e., a "stencil") surrounding a point p. For example, a truncated Gaussian blur convolution [133] only requires points within some hyperrectangle R centered on p. In this case, we can omit non-nearby TLF data when invoking f. This allows for more efficient parallelization.

To better-support parallelization, a developer may optionally include a neighborhood when using a MAP. Formally, given a hyperrectangle stencil R, this MAP overload produces a new TLF defined by:

$$\text{MAP}(L, f, R) = p \mapsto f(p, L_{R+p})$$

A second transformation operator, INTERPOLATE, converts null values into some new value given by a transformation function. Interpolation is useful to "fill in" parts of a TLF that have been discretized due to encoding at a particular resolution. For example, given a nearest-neighbor function nn that gives the color of the closest non-null point in a TLF, the operation INTERPOLATE(L, nn) produces a new TLF with all null values replaced by their closest color. Like MAP, the INTERPOLATE operator offers an overload that accepts a stencil to improve performance.

Finally, the SUBQUERY operator performs an operation on *each partition* in a TLF. For example, Section 3.2.4 shows use of SUBQUERY to encode a TLF's partitions at different qualities.

SUBQUERY logically consists of both SELECT and UNION. Given a subquery q, SUBQUERY executes it over each partition volume $V_1, ...V_n$ in a TLF L. It then unions each partial result into a single output TLF. Formally, it produces a TLF defined as follows:

$$\begin{aligned}&\textsc{subquery}(L, q, m) = \\ &\quad \textsc{union}(q(\textsc{select}(L, V_1)), ..., q(\textsc{select}(L, V_n)), m)\end{aligned}$$

Translation & Rotation. The TRANSLATE operator adjusts every light ray in a TLF by some spatiotemporal distance $(\Delta x, \Delta y, \Delta z, \Delta t)$. Similarly, the ROTATE operator rotates the rays at each point by angles $\Delta\theta$ and $\Delta\phi$. We omit their formal definitions here due to space.

3.2.2 Input & Output

The TLF algebra includes SCAN and STORE operators, which are respectively used to read and write a TLF to LightDB's internal catalog. LOAD and SAVE operators perform equivalent operations to or from a URL (e.g., `file:///input.mp4` or `rtp://localhost`). Finally, developers may optionally use the ENCODE operator to transform video data into a specific encoding (e.g., HEVC); the DECODE operation performs the inverse operation.

Table 3.1: Example expressions using the temporal light field algebra

Description	Algebraic Expression
Self-concatenate a 5-second TLF	$\textsc{union}\Big(\textsc{scan}\left(\text{name}\right), \textsc{translate}\big(\textsc{scan}\left(\text{name}\right), \Delta t = 5\big)\Big)$
Grayscale and H264 encode	$\textsc{encode}\Big(\textsc{map}\big(\textsc{scan}\left(\text{name}\right), \textsc{grayscale}\big), \text{H264}\Big)$
Sharpen middle third of a TLF	$\textsc{subquery}\left(\begin{array}{l}\textsc{partition}\left(\textsc{scan}\left(\text{name}\right), \Delta\phi = {}^{\pi}/_{3}\right), \\ L \mapsto \begin{cases}\textsc{map}(L, \textsc{sharp}) & \text{if } V_\theta(L) = [{}^{\pi}/_{3}, {}^{2\pi}/_{3}] \\ L & \text{otherwise}\end{cases}\end{array}\right)$

3.2.3 Data Definition

CREATE and DROP operators function in a manner equivalent to relational systems. Given a unique name, CREATE creates a new TLF that is a copy of Ω, a distinguished, immutable TLF where each point is associated with the null token ω (see definition 3.1.1). Similarly, the DROP operator removes a TLF from LightDB's internal catalog. Finally, the $\textsc{createindex}(L, d_1, ..., d_n)$ operator is used to create an index over TLF L in dimensions $d_1, ..., d_n$, and the DROPINDEX operator removes a previously-created index.

3.2.4 Examples

TLF algebraic operators can be composed into expressions in the same way as relational algebra operators. Table 3.1 shows a few simple examples. In Chapter 1 we highlighted two VR and AR applications (see Examples 1.1 and 1.2), and now show their implementation in the TLF algebra along with several others:

- **Predictive Spherical Panoramic Video Tiling** (Example 1.1). The predictive tiling application degrades the quality of out-of-view areas of a 360° video in order to improve performance (e.g., up to 75% reduced data transfer). The algorithm in the

Algorithm 3.1: Pseudocode for predictive spherical panoramic video tiling application

inputs : b, bitrate at which to encode high quality region

: n, number of regions to reencode

: $o(t)$, viewer's orientation at time t

output : `out` contains a 360° video reencoded by importance

function *transcode(partition: TLF): TLF*

  if *o(V(partition).t)* $\in$ *V(partition)* **then return** ENCODE *(partition, b)*;

  else return ENCODE *(partition,* $\frac{b}{10}$*)*;

end

let l = LOAD ("input")

let p = PARTITION (l, $\Delta\theta = \frac{2\pi}{n}, \Delta\phi = \frac{\pi}{n}, \Delta t = 1$)

let s = SUBQUERY (p, *transcode*)

SAVE (s, "out")

TLF algebra for Example 1.1 is shown in Algorithm 3.1. This query subdivides the input TLF into one-second segments and partitions of size $(\frac{2\pi}{n}, \frac{\pi}{n})$. Next, the SUBQUERY operator changes the quality of each partition to that given by a function o that predicts a user's future orientation. For example, and as implemented in Algorithm 3.1, we might use dead reckoning to predict a user's next orientation and encode that partition at quality b (and a lower quality $\frac{b}{10}$ elsewhere).

- **Augmented Reality** (Example 1.2). Algorithm 3.2 shows an implementation of Example 1.2 in the TLF algebra. This UDF applies an object detection algorithm such as YOLO9000 [129] and generates a result that is red at detection boundaries and null otherwise. Since YOLO9000 expects an input at a particular input resolution

Algorithm 3.2: An algorithm that performs object detection on an input TLF and overlays bounding boxes around detected objects.

inputs : *detect*, an object detection user-defined function

: n, m, neural network trained resolution

output : `out` contains the input TLF overlaid with bounding boxes

let l = LOAD ("input")

let d = DISCRETIZE (l, $\Delta\theta = \frac{2\pi}{n}, \Delta\phi = \frac{\pi}{m}$)

let b = MAP (d, *detect*)

let u = UNION (l, b)

SAVE (u, "out")

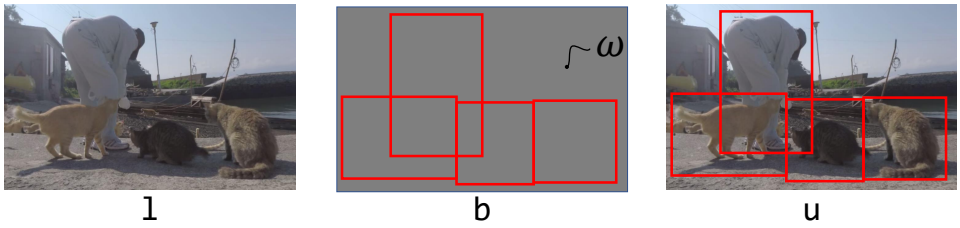


Figure 3.3: Augmented reality TLFs for one 360° frame. Image by Mewpro, CC-BY 3.0.

(416 × 416), the query first lowers the resolution of its input, and applies a MAP using a user-defined function (UDF) called *detect*. Finally, the result is combined with the original input. Figure 3.3 illustrates this process for a single frame of `l`, `d`, and `u`.

- **Depth Map Generation**. Several recent projects have explored real-time depth map generation using parallel or custom hardware [70, 112, 5, 107] in the context of cloud-based VR streaming. For a light field stored in LightDB, the extremely high data sizes (gigabytes or terabytes of raw data per second) make cloud-based streaming to remote clients infeasible. One strategy to reduce the amount of data transfer involves sampling a light field at two points near where a user's eyes are located (i.e., her current position $\boldsymbol{p}$ offset by an interpupillary distance $\boldsymbol{i}$) and computing a depth map for the

Algorithm 3.3: An algorithm to generate stereo depth map information from a TLF containing a light field.

inputs : p, viewer's current position

: i, interpupillary distance

output : `out` contains a stereoscopic 360° video

let in = SCAN ("lightfield")

let l = SELECT (in, $p - \frac{i}{2}$)

let r = SELECT (in, $p + \frac{i}{2}$)

let s = UNION (l, r)

let i = INTERPOLATE (stereo, depth)

SAVE (i, "out")

360° videos incident to those points [88]. An implementation of this query in the TLF algebra is shown in Algorithm 3.3, where the `depth` function implements the logic described in [107].

- **Non-VR & AR applications**. Although the TLF algebra was designed for VR and AR applications, it may also be used to express general VWAs. For example, the non-VR application described in Example 1.3 may be implemented using the query shown in Algorithm 3.4, which assembles videos within a region R into a mosaic using a "horizontal stacking function" *hstack*. Chapter 5 shows further examples of VWA queries expressed in the TLF algebra (see Tables 5.2 to 5.5).

3.3 Query Language

LightDB includes a declarative query language called *VRQL*. VRQL is an implementation of the TLF algebra described in the previous section. It allows developers to describe queries without the need to be concerned with the underlying complexities of the video data, how

Algorithm 3.4: An algorithm that creates a mosaic out of one or more surveillance cameras in a region.

inputs : $c_1, ..., c_n$, 2D surveillance cameras

: R, spatial region of interest

output : `out` contains a mosaic of the cameras falling in the given region of interest

let c = UNION (SCAN (c_1), UNION (SCAN (c_2), ...))

let n = SELECT (c, R)

let m = MAP (n, *hstack*, V(n))

SAVE (m, "out")

the query is executed, or which hardware to use for individual operations. This abstraction is a key distinguishing feature from existing video processing systems, which require developers to manually manage these details. We currently have bindings for VRQL in C++ and Python. Our examples in this section show use of the Python variant.

In VRQL, developers write queries over TLFs using functions that correspond to the TLF algebra. As a concrete example, consider the following example application, where a developer wishes to perform the following operations live 360° video in order to stream it adaptively:

i Ingest a live video from a 360° camera rig

ii Overlay a watermark on the video

iii Apply an image sharpening convolution

iv Temporally partition the intermediate result into short segments to support adaptive streaming

v Encode each segment into a format suitable for the available network bandwidth of a client device

To execute this example application, a developer writes the following Python query (see Figure 3.5 for the equivalent algebraic structure):

```
Load("rtp://localhost/input")
 .Union(Scan("W"))
 .Map(sharpen)                                                    (3.2)
 .Partition(Time, 2)
 .Subquery(lambda p: p.Encode(Codec.H264))
 .Save("output.mp4")
```

LightDB exposes each operator introduced in Section 3.2 as VRQL functions. As is common in modern languages, the fluent interface $f(\alpha).g(\beta)$ is used as shorthand for composition $g(f(\alpha), \beta)$.

To improve readability, a VRQL query may assign an intermediate result to a variable. For example, the following query is equivalent to the TLF concatenation example shown in Table 3.1:

```
auto tlf = Scan(name)
auto concat = tlf.Union(tlf.Translate(Time, 5))
```

LightDB also allows developers to create indices over existing TLFs using the CREATEINDEX method. For example, the following query modifies `cat` from the previous example by selecting only the first three seconds of video data (line 1). Line 2 then creates an index over two spatiotemporal dimensions of `out`. LightDB's query optimizer may then elect to utilize this index on line 3.

```
cat.Select(Time(0, 3)).Store("out")
Scan("out").CreateIndex(Y, Time)
Scan("out").Select(Y(0, 0), Time(0, 1)).Map(grayscale)
```

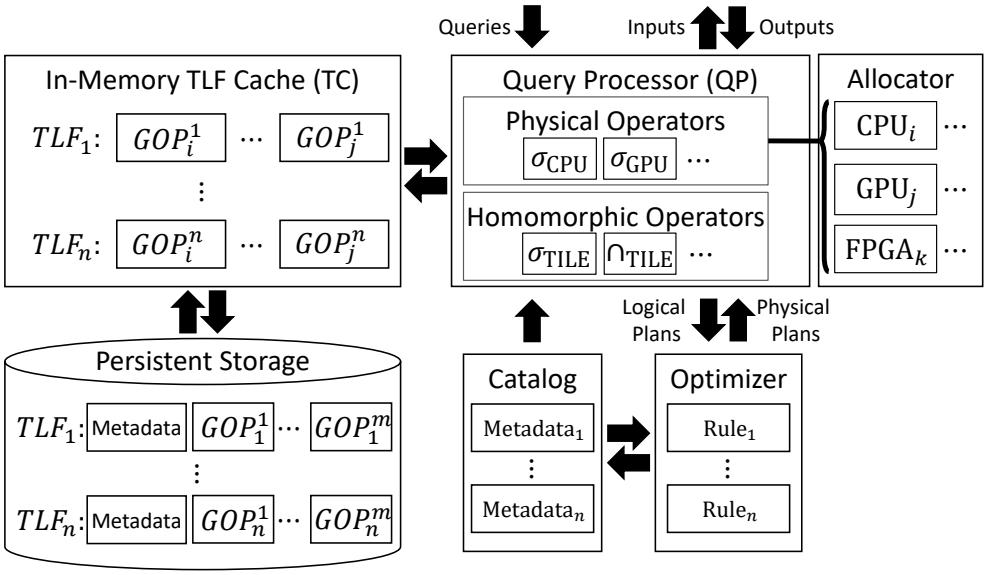


Figure 3.4: LightDB architecture

Finally, LightDB supports user-defined functions (UDFs) that may be used with the MAP and UNION operators for functionality not available in its built-in library. For example, to create the TLF shown in Figure 3.2(a), a developer would define an anonymous function used in MAP in the following query:

```
x = 1
tlf = Create("RB")
  .Select(Volume((-x, x), ...))
  .Map(lambda p: Color.Red if p.volume.x < 0 else Color.Blue)
```

3.4 Architecture

In this section, we present LightDB's current overall architecture together with the details of its core components. LightDB is currently a single-node, primarily single-threaded system written in C++, and contains approximately 20,000 lines of code.

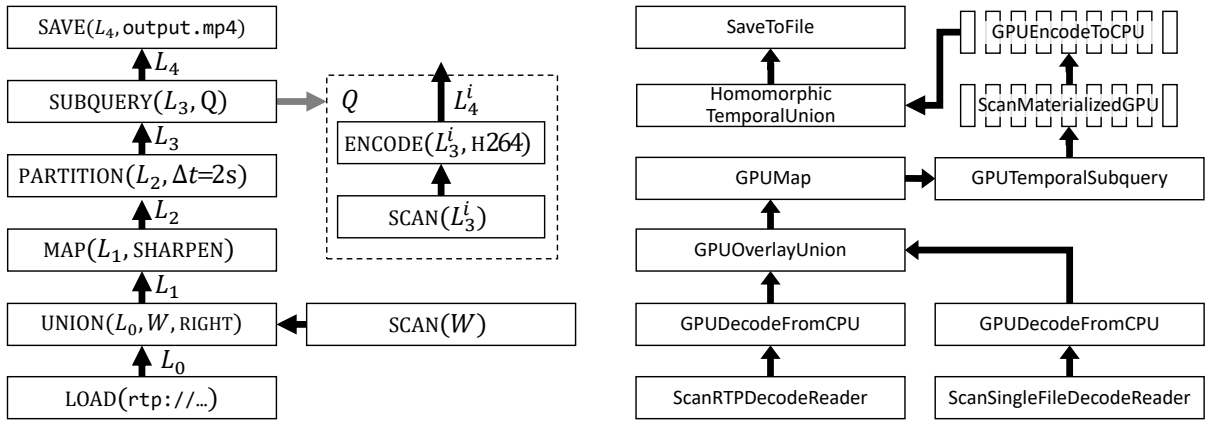
In LightDB, users submit individual queries as a statement or script that may include variable assignments. Writes to TLFs are versioned, and version numbers are stored as part of the TLF's metadata (see Section 3.4.4). LightDB uses the immutability and versioning of TLFs to provide snapshot isolation during query evaluation. When a query references a TLF, LightDB operates on the most recent version available. Developers may optionally parameterize the SCAN operator with a version number. We currently disallow queries that overwrite the same TLF more than once.

LightDB supports both one-shot and streaming queries, with each executed similarly. Either query type may operate over TLFs that are being continuously ingested (i.e., they have their streaming flag set), those already stored in LightDB, or from other data sources such as a socket, local disk, or distributed file system. All of the operators in LightDB are non-blocking, though user-defined functions used as arguments may lead to blocking.

In our prototype, TLFs are immutable and writes are performed by persisting video data at the track granularity (see Chapter 2). For example, if a query overwrites a TLF after modifying information visible in video track T, LightDB materializes an updated version T' and writes it to disk alongside T. Unmodified tracks are not rewritten; LightDB instead stores pointers to the original video tracks.

The major components of LightDB are shown in Figure 3.4. The Query Processor (QP) receives declarative queries as input. It converts them into physical query plans that it executes and returns results to applications in the form of encoded videos.

During query execution, the (QP) interacts with an in-memory TLF cache (TC) to hold data loaded from persistent storage. As shown in Figure 3.4, the TC contains entries for TLF catalog metadata entries, which are parsed from their MP4 representation prior to caching. It also contains a buffer pool for encoded (see Chapter 2) and contiguous sequences of decoded frames that have been recently accessed. Buffering at this granularity improves temporal locality and reduces misses for predictive frame requests. The TC uses reference counting and a least-recently used eviction policy.



(a) Logical Plan

(b) Physical Plan

Figure 3.5: LightDB logical and physical plans for the query shown in Equation 3.2, which watermarks, sharpens, and adaptively encodes a 360° video.

The translation of the input declarative queries into logical query plans is a straightforward one-to-one mapping. The logical-to-physical query plan translation, however, is amenable to various optimizations that we describe in the following section.

3.4.1 Query Optimization

The LightDB algebra is amenable to several optimizations that improve performance. To demonstrate this, LightDB comprises a rule-based query optimizer (QO) that performs two types of optimizations.

The first type of optimization involves selecting the physical implementation for each logical operation, including the device that should execute the operation. The selection is heuristic and proceeds in a bottom-up fashion. We also have rules that combine, reorder, and eliminate operators. We describe each in this section.

Given a logical query plan as input, the QO generates the physical plan by first transforming the logical plan in a bottom-up fashion starting from the SCAN and DECODE operations. During that transformation, the QO uses two heuristics: (1) GPU-based operators are faster than FPGA-based ones, which in turn are faster than CPU-based ones and (2) it is more efficient to keep data on the same device for consecutive operations.

The QO first selects decoders, which are leaves in the query plan graph. For each TLF stored within LightDB, the QO consults the TLF catalog (TC) and selects a GPU-based SCAN physical operator if one exists for the video codec; otherwise it uses a CPU-based SCAN implementation. For TLFs ingested using a DECODE operator, the QO first checks for user-supplied hints (e.g., DECODE(`url`, HEVC)). If no hint is supplied, the QO attempts to infer a codec by examining (in order) MP4 metadata, the file extension, or a prefix of the input data. If the QO is unable to identify a codec, query processing fails. Otherwise, the QO uses a GPU-based decoder if one is available for a codec and falls back to a CPU implementation if none exists.

Having selected physical operators for the leaves in the query plan, the QO then selects physical operators for the remaining operator nodes in a bottom-up, breadth-first manner. For each unary operator, the QO selects a physical operator that executes on the same device as its predecessor. If no physical operator implementation is available for that device, the QO selects a GPU-based implementation and inserts a TRANSFER operator to mediate the inter-device transfer. User-defined functions used in MAP and other operators, which may include implementations that target CPUs, GPUs, or FPGAs, are similarly mapped to physical operators.

A similar process applies to the n-ary UNION operator. If all predecessors execute on the same device, the union also executes on that device. Otherwise, a GPU-based operator is selected and TRANSFER operators are inserted.

A concrete example of this process is shown in Figure 3.5. On the left-hand side is a logical plan constructed from the query labeled Equation 3.2 from Section 3.3. The right-hand side shows one possible physical execution plan produced after the QO executes the steps described above.

After producing an initial physical query plan, the QO performs further optimizations as follows. First, it makes the following transformations and eliminations, which are applied in order, iteratively, and until fixpoint:

- Consolidate consecutive MAPs: $\text{MAP}(\text{MAP}(L, f), g) \rightarrow \text{MAP}(L, f \circ g)$ when both are executed on the same device.
- Remove redundant selections: $\text{SELECT}(\text{SELECT}(L, [d_1, d_1'])), [d_2, d_2']) \rightarrow \text{SELECT}(L, [d_2, d_2'])$ if $d_1 \leq d_2$ and $d_1' \geq d_2'$.
- Combine partitions and discretizations: $\text{PARTITION}(\text{PARTITION}(L, \Delta d = \gamma), \Delta d = \gamma') \rightarrow \text{PARTITION}(L, \gamma')$, and $\text{DISCRETIZE}(\text{DISCRETIZE}, \Delta d = \gamma), \Delta d = \gamma') \rightarrow \text{DISCRETIZE}(L, \gamma')$, if $\gamma' = i \cdot \gamma$, where $i \in \mathbb{Z}$.
- Convert $\text{DISCRETIZE}(\text{INTERPOLATE}(L, f), \Delta d = \gamma) \rightarrow \text{MAP}(L, \mathcal{D}(f))$, where D is a function that produces f on discretization intervals and null otherwise.
- Combine interpolations and maps: $\text{INTERPOLATE}(\text{MAP}(L, f), g) \rightarrow \text{INTERPOLATE}(L, f \circ g)$.

Second, the QO "pushes up" instances of the INTERPOLATE operator. Delaying interpolation ensures that a TLF remains discrete for as many operations as possible. The QO currently pushes interpolation above SELECT and PARTITION operators. Moving interpolation may further eliminate operators as described above.

Finally, the QO attempts to substitute highly efficient homomorphic operators (HOps) that may be executed directly on encoded TLF video. For example, unioning non-overlapping TLFs can often be performed using a homomorphic operator. Because video encoding and decoding is expensive relative to most other operations (see Figure 3.7), HOps always outperform operators that execute over decoded video (by up to 500×; see Figure 3.11), even if they require inter-device transfer.

3.4.2 Physical Algebra

Our current implementation includes physical operator variants that target CPUs, GPUs, and FPGAs and communicate using PCIe.

First, LightDB has a CPU-based implementation for each of the operators described in Section 3.2. These operators rely on FFmpeg [17] for video encoding and decoding and make direct modifications to decoded video frames.

LightDB also includes a GPU-accelerated operator implementation for each logical operator. The ENCODE and DECODE GPU-based operators utilize the hardware-accelerated NVENCODE/DECODE interfaces [113]. The remaining operators each use CUDA [114] kernels to perform their work.

LightDB also has homomorphic operators (HOps) that perform operations directly on encoded video data *without requiring that it be decoded*. This leads to higher performance — up to 500× faster compared to GPU-based operators (see Figure 3.11). Two categories of HOps exist in LightDB: those that operate over encoded groups of pictures (GOPs; see Chapter 2), and those that operate over the tiles (ibid.) in each frame. These operators are currently executed on the CPU, and we plan on adding other HOps (e.g., keyframe selection, scalable video coding [136]).

To understand the first category, consider a video encoded with a GOP duration of one second. The GOPSELECT operator may be applied for any SELECT operator that temporally selects precisely at a GOP boundary (e.g., SELECT$(L, t = [i, j])$, where $i, j \in \mathbb{Z}$). It does so by using the GOP index to identify the byte region in an encoded video file that contains the frame data for the relevant GOPs, and outputs them *without decoding*.

The GOPUNION operator performs a similar operation—given n encoded videos that are temporally-contiguous, it concatenates the encoded GOPs in each video and produces a valid unioned result.

The second category of HOps perform a similar operation over the tiles (q.v. Chapter 2) within each video frame. The tileSelect operator may be used for any angular selection that includes complete, contiguous tiles. It does so by using tile index, as shown in Figure 2.3(b), to efficiently identify the relevant bytes without video decoding. For example, consider a video that has been tiled as shown in Figure 2.3(a). The tileSelect operator may be used for the logical selection $T_1 = \text{SELECT}(\phi = [0, \frac{\pi}{2}))$, since it precisely selects the tile labeled 1 in Figure 2.3(b). Similarly, $T_{23} = \text{SELECT}(\phi = [\frac{\pi}{2}, \pi))$, selects tiles labeled 2 and 3.

Analogous to GOP unioning, the tileUnion HOp concatenates tiles without video decoding. To be applicable, each UNION operand must be defined at the same spatiotemporal points and the tiles must be angularly non-overlapping. For example, in Figure 2.3(a) the operation $\text{UNION}(T_1, T_{23})$ can be performed using this operator.

3.4.3 Physical Organization

In Chapter 2 we introduced two common encoding methods for virtual and augmented reality video data: spherical panoramas (i.e., 360° videos optionally containing depth information) and light slabs. The first method efficiently represents visual data incident to a *point*, while the last is efficient at representing visual data incident to a *plane*.

LightDB supports both methods by physically representing each TLF using one of two physical formats: a *physical point TLF (PointTLF)* or a *physical light slab TLF (PlaneTLF)*. Each PointTLF contains one or more 360° videos, with a separate video stream encoded for each non-null spatial point in the TLF. PlaneTLFs contain one or more slabs at various positions and orientations, with a separate stream encoded for each.

Both PointTLFs and PlaneTLFs may be continuous or discrete. The video data associated with a discrete TLF is materialized and encoded into a video stream. For a continuous TLF, since it is not possible to materialize every point in a volume of continuous space, LightDB instead creates a partially materialized view by materializing an intermediate TLF up to the latest point where it becomes continuous (i.e., the last INTERPOLATE operator). It encodes this intermediate result as if it were an ordinary, discrete TLF. It identifies

the remaining logical operator subgraph that acts on the intermediate result to produce the full (continuous) query result. This subgraph is serialized in a special *view subgraph* field alongside other TLF metadata. For example, given a discrete TLF L and the query INTERPOLATE(SCAN(L), f), which produces a continuous TLF, LightDB materializes L and records the call to INTERPOLATE and f in the view subgraph.

Combinations of PointTLF and PlaneTLFs may be merged through application of the UNION operator. Similar to the above, LightDB materializes and stores each of the inputs to the union and records the union operator in the view subgraph field in the TLF metadata. The resulting *composite TLF (CompositeTLF)* contains any number of child PointTLFs and PlaneTLFs, potentially recursively.

During encoding and when persisting, LightDB automatically converts between TLF formats using the following rules:

- A PlaneTLF is converted to a PointTLF whenever it is spatially selected at a single point or spatially discretized to a small number of points (≤ 4 in our prototype).
- A CompositeTLF is transformed to a PointTLF whenever it contains only PointTLFs, and to a PlaneTLF when it contains only PlaneTLFs.
- When a PointTLF or PlaneTLF is no longer contained within its bounding volume, it is dropped.

3.4.4 Data Storage

We now describe how discrete PointTLFs and PlaneTLFs are physically stored on disk. First, observe that a PointTLF contains one or more encoded 360° videos defined at spatially distinct points. As described in Chapter 2, each encoded video is associated with a projection function (which defines how the visual data is projected onto a frame), a data stream compressed using a video codec, and an optional depth map metadata stream.

Similarly, a PlaneTLF contains a set of light slabs (defined in Section 2.3.2), where each slab is associated with a data stream compressed using a video codec, geometric metadata that includes the start and endpoints of the planes in three-dimensional space, and sampling parameters (i.e., the number of *uv* and *st* plane samples).

For each of the formats, LightDB physically organizes a TLF within a single directory on the file system. LightDB uses a multi-version, no-overwrite mechanism for TLF writes, and each directory contains one *metadata file* for each version of the associated TLF. The metadata file is a small (generally less than 20kB) MP4-compliant multimedia container (see Section 2.2) that contains information about each version of a physical TLF. Using the MP4 format allows for better interoperation with viewing headsets and video processing libraries.

The metadata MP4 file is composed of a forest of data elements called atoms (see Section 2.2) that contain the properties of the TLF and pointers to associated video streams. When a new TLF version is created through a STORE, LightDB increments the version number and atomically creates a new metadata file containing information about the new version (i.e., `5-metadata.mp4` corresponds to TLF version five).

The directory also contains one or more *video files* containing encoded video streams. This structure allows multiple TLFs to maintain pointers to the same encoded video files and avoids data duplication. This allows multiple TLFs to reference the same video file without requiring it to be duplicated on disk.

As described in Chapter 2 and illustrated in Figure 2.4, LightDB uses standard MP4 atoms to store pointers to separately-stored encoded video streams and the `sv3d` drawn from the Spherical Video V2 RFC [141] to store a PointTLF's projection function.

LightDB extends the MP4 format by introducing an additional atom (`tlfd`) to serialize the remaining data about a TLF's physical type. For a PointTLF, this includes the points at which the TLF is defined. For a PlaneTLF, this includes geometry and sampling granularity for each of the slabs. Finally, a CompositeTLF recursively contains two or more `tlfd` atoms as children.

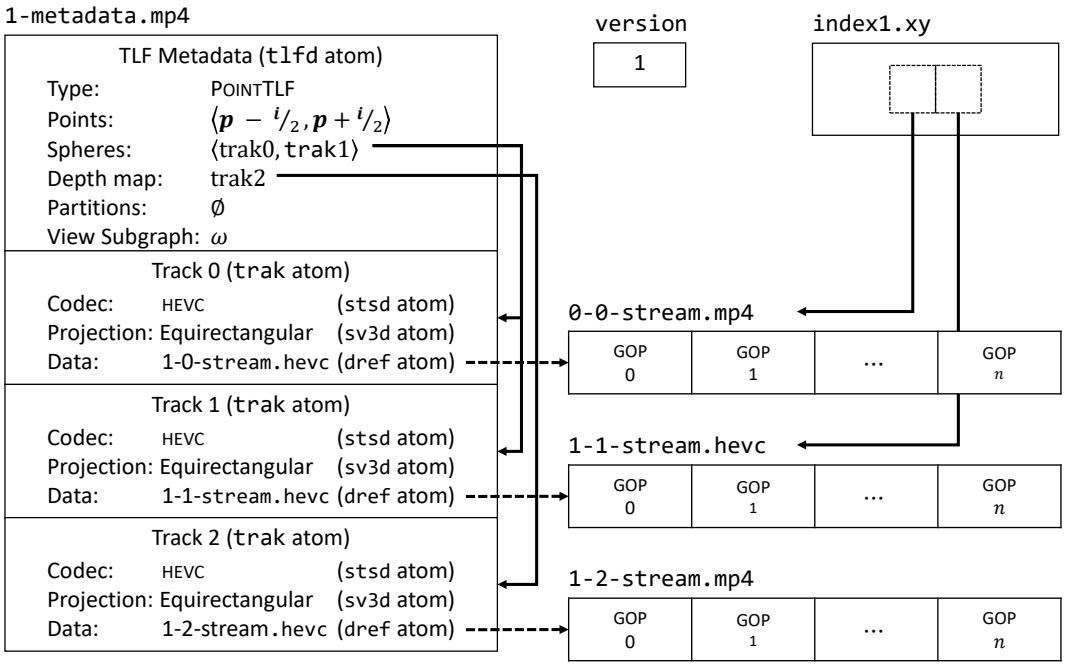


Figure 3.6: Physical layout of a TLF defined at two points with depth metadata.

Each `tlfd` atom also contains pointers to video tracks, which store metadata about the underlying video data that support the discrete TLF (see Chapter 2). For a PointTLF, the `tlfd` atom contains a pointer to one or more video spheres and optional pointers to depth map tracks. PlaneTLFs contains one pointer per slab.

To illustrate this structure, we show in Figure 3.6 an abridged physical layout for a TLF defined at the origin. This TLF's current version (stored in a file named `version`) is one, and its associated metadata file `1-metadata.mp4`. This file contains a `tlfd` atom that describes the TLF. It is discrete, and so has no view subgraph. It has a pointer to a single `trak` video track that contains the codec used to encode the video data, the projection function, an index to the beginning of each GOP within the video file, and a pointer to the file that contains the encoded video data: i-j`-stream.mp4` (i.e., version i, stream j).

3.4.5 Indexing

In Chapter 2, we introduced two indices found in encoded video: a tile index (Figure 2.3) and a GOP index (Figure 2.4). When available, the LightDB QO uses the former for point or range queries over one or more angular dimensions, and the latter for point or range queries over time.

LightDB also supports *spatial indices*, which are external indices over any combination of spatial dimensions and take the form of R-trees [59] that identify relevant encoded video files in a TLF. Such indices are useful in the case of a TLF created from the union of videos or light fields captured at different locations, such as at a concert, museum, or tourist location.

When executing a query, the QO first chooses a spatial index (if any) that covers the largest subset of spatial dimensions included in each selection. For example, if a user has executed the command CREATEINDEX(L, x, z), LightDB utilizes the resulting R-tree for queries of the form SELECT$(L, x \in [a, b], z \in [c, d], ...)$. As shown in Figure 3.6, a spatial index is stored as an external file with a name containing its version and covered dimensions (e.g., `1-index.xz`).

After the QO considers spatial indices (if any), if a temporal constraint is present in a selection, LightDB uses GOP indices (if present) to identify relevant temporal regions in an encoded video file. Each GOP index is embedded in the `stss` region of a TLF's metadata (see Figure 2.4) and maps a starting time to the byte offset of the associated GOP. Given a temporal selection (e.g., SELECT$(t \in [a, b], ...)$), the QO uses this information to look up GOPs containing information between time a and b.

Finally, the QO considers a tile index (if present) to identify applicable and independently-decodable subregions of each frame. For example, assuming an equirectangular projection, a query of the form SELECT$(\phi \in [0, \frac{\pi}{2}])$ might only need to decode Tile 1 shown in Figure 2.3. Since this index is also used by video decoders, an attempt to drop an angular index results in an error.

3.5 Evaluation

We have implemented a prototype of LightDB in C++ using ∼20,000 lines of code. GPU-based operators were implemented using NVENCODE/NVDECODE [113] and CUDA 8.0 [114]. We experimentally evaluate LightDB and compare it to four baseline systems in terms of programmability for complex VR video workloads (Section 3.5.1) and performance (3.5.2). In Section 3.5.3 we show how LightDB is able to utilize hardware accelerators, and in Section 3.5.4 we detail LightDB operator performance.

Baseline systems. We compare LightDB against OpenCV 3.3.0 [118] and FFmpeg 3.2.4 [17], the most commonly-used frameworks for 2D video processing and analysis. OpenCV is a computer vision library designed for computational efficiency and high-performance image analytics, while FFmpeg is a platform designed for video processing that is invoked via the command-line interface (CLI) or by linking to its C-based API.

We build both FFmpeg and OpenCV with support for GPU optimizations. FFmpeg is configured with support for NVENCODE/NVDECODE [113] GPU-based encoding and decoding. OpenCV is built with internal calls to FFmpeg and CUDA 8.0 [114].

We also compare against Scanner [124], a recent system designed to efficiently perform video processing at scale. We installed Scanner by using its most recently-published Docker container. This GPU-enabled container was built using Ubuntu 16.04, OpenCV 3.2, CUDA 8.0, and FFmpeg 3.3.1.

360° videos are loaded into OpenCV, FFmpeg, and Scanner as encoded, two-dimensional equirectangular projections of a video sphere. While none of these systems natively support operations on light slabs, in some cases we were able to apply operations directly on encoded SlabTLF videos (e.g., conversion to grayscale). For other light field operations that are not readily supported by the comparison systems, we show only LightDB results.

Finally, we compare against SciDB 15.12 [22, 131], which is a distributed, array-oriented database management system designed for efficient array processing at scale. For experiments involving SciDB, we represent 360° videos as a non-overlapping decoded three-dimensional array (x, y, t) and light fields as discrete six-dimensional arrays encoded as shown in Figure 2.7(b).

Experimental configuration. We perform all experiments using a single node running Ubuntu 14.04 and containing an Intel i7-6800K processor (3.4 Ghz, 6 cores, 15 MB cache), 32 GB DDR4 RAM at 2133 MHz, a 256 GB SSD drive (ext4 file system), and a Nvidia P5000 GPU with two discrete NVENCODE chipsets.

Datasets. We use the following two reference datasets in our experiments, one for 360TLFs and another for SlabTLFs. For 360TLF experiments, we utilize the "Timelapse", "Venice", and "Coaster" videos from Corbillon et al [32]. Each of these 360° videos are equirectangularly projected, 142-177 MB in total size, captured at 30 frames per second at 4K resolution (3840×2048), and have an average bit rate of 13–15Mbps. Except for Scanner, we truncated each video to the same duration (90 seconds) with one-second GOPs. As detailed below, Scanner was unable to complete queries for these inputs, and so we show its results for a further-abbreviated 20-second variant.

For SlabTLFs, we use the "Cats" light slab by Wang et al [155]. This light slab is encoded at 4096×2816 resolution with 8×8 uv-plane samples. Since this dataset is provided as 109 separate images, we converted it into a video using the H264 codec at 30 frames per second with one-second GOPs. We also looped the frames so that the resulting slab had a total duration of 90 seconds (for LightDB, OpenCV, and FFmpeg) and 20 seconds (Scanner).

3.5.1 Programmability

To evaluate the programmability of LightDB relative to similar systems, we execute VRQL queries for the predictive 360° tiling and augmented reality (AR) workloads described in Section 3.2.4.

For the predictive 360° tiling application, LightDB loads each 360° dataset (Timelapse, Venice, and Coaster) from the file system, decodes it as a 90-second 360TLF, and partitions it into one-second fragments. It then decomposes each partition into sixteen equally-sized tiles ($\Delta\theta{=}\frac{\pi}{2}, \Delta\phi{=}\frac{\pi}{4}$) and re-encodes one tile at high quality (HEVC at 50Mbps) and the remaining fifteen at low quality (50kbps). It finally recombines the tiles and writes the result to the file system. This process is repeated for each of the 90 one-second fragments in the input. To emulate looking in different directions, the high quality tile is initially the upper-left of the equirectangular projection and advanced in raster order (modulo 16) every second.

For the AR application, LightDB loads each dataset from the file system and decodes it into a 360TLF (for Timelapse, Venice, and Coaster inputs) or SlabTLF (Cats dataset). This TLF is then discretized and fed into a UDF that executes the YOLO9000 detection algorithm [129]. This result is finally unioned with the original TLF.

For the SciDB, OpenCV, FFmpeg, and Scanner variants, we use the same inputs and map each step into a system-specific equivalent. Both OpenCV applications are written in C++. For FFmpeg, we execute the predictive 360° tiling workflow using both a C++ implementation and via the command-line interface (CLI). The FFmpeg CLI does not expose extensibility for custom UDFs, so we did not execute the AR application for this case. We used the Scanner Python API and leveraged existing functions when possible and implemented custom kernels using its C++ API for unsupported operations (e.g., tiling, recombining). Since SciDB does not natively expose video-related functionality, we implement encode, decode, and the AR UDF externally using OpenCV and transfer data using standard SciDB import and export operations.

We show the number of lines of code associated with each query and system in Table 3.2. Here, both LightDB and SciDB are able to express the complex workload in a small number of lines, while OpenCV and FFmpeg require many more lines of code to express the same query. The Scanner versions fall between these extremes.

Table 3.2: Lines of code required to reproduce the predictive 360° and augmented reality queries described in Section 3.2.4. Numbers in parenthesis show the lines required to implement user-defined functions for operations not natively supported.

System	Lines of Code	
	360° Tiling	Augmented Reality (UDF)
LightDB	9	9 (18)
SciDB	12	13 (98)
Scanner	37 (144)	35 (110)
OpenCV	112	90
FFmpeg	283	161
FFmpeg CLI	895	N/A

The VRQL and SciDB queries for both of these workloads are declarative, with a developer needing only to express the form of the desired result. By contrast, the implementation for OpenCV and FFmpeg is much more imperative, and developers need to be heavily involved in deciding *how* the workload is executed. This includes details such as calculating 2D tiling, copying data between video frames, calculating codec parameters, and managing file IO. Scanner again falls between these two extremes, where a developer must decide some execution details (e.g., selecting hardware, calculating tile sizes, aligning unsynchronized frame rates) but is spared from others. For each comparison, managing these details requires additional lines of code that are interspersed with application logic.

For these workloads, LightDB produces correct results using significantly fewer lines of code than the imperatively-oriented video frameworks, and is able to do so in a declarative manner that avoids requiring developers to be involved with the low-level details associated with workload execution.

3.5.2 Application Performance

We next evaluate the performance of LightDB and compare it to OpenCV, FFmpeg, Scanner, and SciDB. For this comparison, we evaluate the performance of the applications described in the previous subsection: predictive 360° tiling and augmented reality.

Predictive 360° Tiling. We first execute the predictive 360° tiling application described in Section 3.2.4 using the Timelapse, Venice, and Coaster datasets.

The throughput for each system and dataset is shown on the left plot of Figure 3.7(a). Here, LightDB is able to process up to 4×, 7×, 13×, and 190× the number of frames per second compared to the FFmpeg, OpenCV, Scanner, and SciDB systems, respectively. SciDB suffers due to its lack of native video encoding support, which necessitates an expensive export/import cycle to/from an external UDF. On the other hand, Scanner pins *all* uncompressed frames in memory and requires an expensive per-tile, per-frame allocation. This quickly exhausts available memory and prevents operations on video that cannot be completely materialized (e.g., 4K videos longer than ∼20 seconds).

The key means by which LightDB is able to achieve the highest performance is that it utilizes its efficient physical tile union operator (TILEUNION, see Section 3.4.2) that avoids an expensive additional decode/encode step that is required by the other systems. At runtime and as discussed in Section 3.4.1, LightDB recognizes that this physical tile union HOp is applicable and automatically uses it.

This ability to automatically select from many available optimizations is a key strength for LightDB. Its declarative VRQL language removes the need for a developer to hard-code the low-level mechanics of query execution, which allows LightDB to *automatically* apply available optimizations at runtime.

The predictive tiling workload is motivated by related work [67, 56, 48, 106], which demonstrates a substantial decrease in video size by encoding regions of a 360° video at different qualities. We thus evaluate LightDB and the baseline systems on their ability

to decrease total 360° video size. Table 3.3 shows the results. LightDB is able to offer performance comparable to FFmpeg. Here, the other systems (which all depend on OpenCV in our experimental configuration) suffer due to their lack of robust support for codec settings.

Finally, the right graph in Figure 3.7(a) breaks down total query execution time by LightDB operator for the Timelapse dataset and various tile configurations. Across each tile configuration, total execution time is dominated by the GPU-based encode and decode and not by other query operators such as UNION and MAP.

Augmented Reality (AR). We next execute the AR application described in Section 3.2.4 using the 360° datasets as input.

Throughputs for LightDB and comparison systems are shown on the left plot in Figure 3.7(b). Here LightDB is again able to process up to 21× more frames per second than OpenCV, 3× for FFmpeg, and 8× for Scanner. This performance is possible because LightDB is able to perform most of the processing (decode, discretize, and union) using its GPU-optimized physical operators and parallelize the GPU-to-CPU transfer required by the UDF. Neither OpenCV nor FFmpeg are able to fully parallelize the operation, and OpenCV suffers in particular due to its lack of support for NVENCODE on Linux. After reviewing Scanner's internal implementation, we observed that Scanner's performance is degraded because it relies on OpenCV to convert frames to a format compatible with the object detection algorithm. Additionally, Scanner's built-in bounding box overlay operator also relies on OpenCV.

The right plot in Figure 3.7(b) shows each operator's contribution to query execution time for each dataset. As before, the GPU-based encode and decode operations account for a substantial portion of total execution time. The object recognition UDF, which requires a GPU-to-CPU transfer, constitutes the bulk of the remaining time.

3.5.3 Hardware Acceleration

An important feature of LightDB is its ability to integrate specialized hardware accelerators in its query execution pipeline. To demonstrate this ability and to illustrate its performance potential, we characterize the performance for the depth map generation application described in Section 3.2.4 using a UDF with a CPU and hybrid FPGA implementation [107] executed on a Xilinx Kintex-7.

For this experiment, we show results for the Cats SlabTLF (sampled at two spatial points) and the Timelapse 360TLF experiment using adjacent frames. Figure 3.8 compares performance for CPU and hybrid FPGA versions of the query. Here introducing a FPGA-based UDF variant allows the LightDB query optimizer to reduce query execution by more than 25%. This is a useful performance advantage since high-quality depth map generation is computationally expensive and is often performed offline [107].

3.5.4 Operator Performance

360TLF Operator Performance. We first examine the individual performance of LightDB operators for 360TLFs using the Timelapse dataset. Figures 3.9(a)–(d) respectively illustrate the operator performance for the SELECT, MAP, UNION, and PARTITION operators. For each unary operator O, we executed a minimal query **Decode**(L).O(...).**Store**(L').

To benchmark the selection and partition operators, we select a subset of or partition the 360TLF along different dimensions.

For the MAP operation, we use a grayscale UDF that drops the chroma signal from its input and a blur UDF that performs a truncated Gaussian blur convolution [133].

We demonstrate the UNION operator by combining the Timelapse dataset with three TLFs. We first UNION it with the Venice dataset. Next, we use a 360TLF that contains a 64×64 watermark; this is denoted as "Watermark" in Figure 3.9. Finally, the "Rotated Timelapse" TLF contains the original Timelapse dataset rotated by 90°. All operations use the LAST merge function (see Section 3.2).

For each operator shown in Figures 3.9(a)–(d), LightDB outperforms other systems by a modest amount. This result is expected, since application of each operator requires the same expensive decode/encode cycle that dominated execution time in our previous experiments. For most operations, LightDB performs better since it utilizes its GPU-based physical operators across the entire query, which minimizes data transfer. The one exception is with temporal selection and partitioning, where LightDB outperforms the other systems by a sizable margin. In this case LightDB uses its GOP index to decode only the relevant portions of the 360TLF. LightDB performs slightly worse for unions due to its merge UDF overhead.

SlabTLF Operator Performance. We next show the average throughput for queries using SlabTLFs as input. Figure 3.10 shows the results for the SELECT and MAP operators using various input parameters. Because the baseline systems do not support light fields, we only show results for LightDB.

For the SELECT operator, we show in Figure 3.10(a) selection at one or two points (representing a mono or stereoscopic selections), over the temporal interval $t = [1^1/_2, 3^1/_2]$, and over angles. Here LightDB is able to generate results at approximately 60 frames per second, which is a common throughput for 4K VR video. Similarly, blur and grayscale operations, shown on Figure 3.10(b), perform at a rate comparable to their 360TLF counterparts (see Figure 3.9(b)).

Effectiveness of Optimizations. Finally, we evaluate the performance impact of several LightDB optimizations in Figure 3.11.

The left two plots in Figures 3.11(a) and 3.11(b) show performance of the HOps in LightDB, in terms of the application's frames per second. These operators allow LightDB to far outperform the baseline systems—in some cases by more than 500×! The one exception is FFmpeg's GOP unioning performance, which matches LightDB because it utilizes a similar GOP stitching mechanism that it calls a "concat protocol" [30].

The "Self Union" plot illustrates the effect of other query optimizations on LightDB performance. Here, we execute the degenerate query $\textsc{union}(L, L)$, which LightDB simplifies to L to produce a result without an expensive decode. The other systems are unable to recognize this pattern and perform far more work to produce the degenerate result. A similar effect is seen in the "Self Select" plot, where we show results for the degenerate $\textsc{select}(L, [-\infty, +\infty])$.

3.5.5 Index Performance

Our final evaluation explores index performance in LightDB using queries of the form **`Scan(`**L**`).Select(`**$d_i \in [a, b]$**`).Store(`**L'**`)`**, where d_i is an indexed dimension.

Figure 3.12(a) shows performance of temporal selection on the Timelapse dataset for two choices of $[a, b]$ both with and without a GOP index. Here the presence of the index substantially improves performance for queries over a small interval but does not impact performance for queries over a large extent.

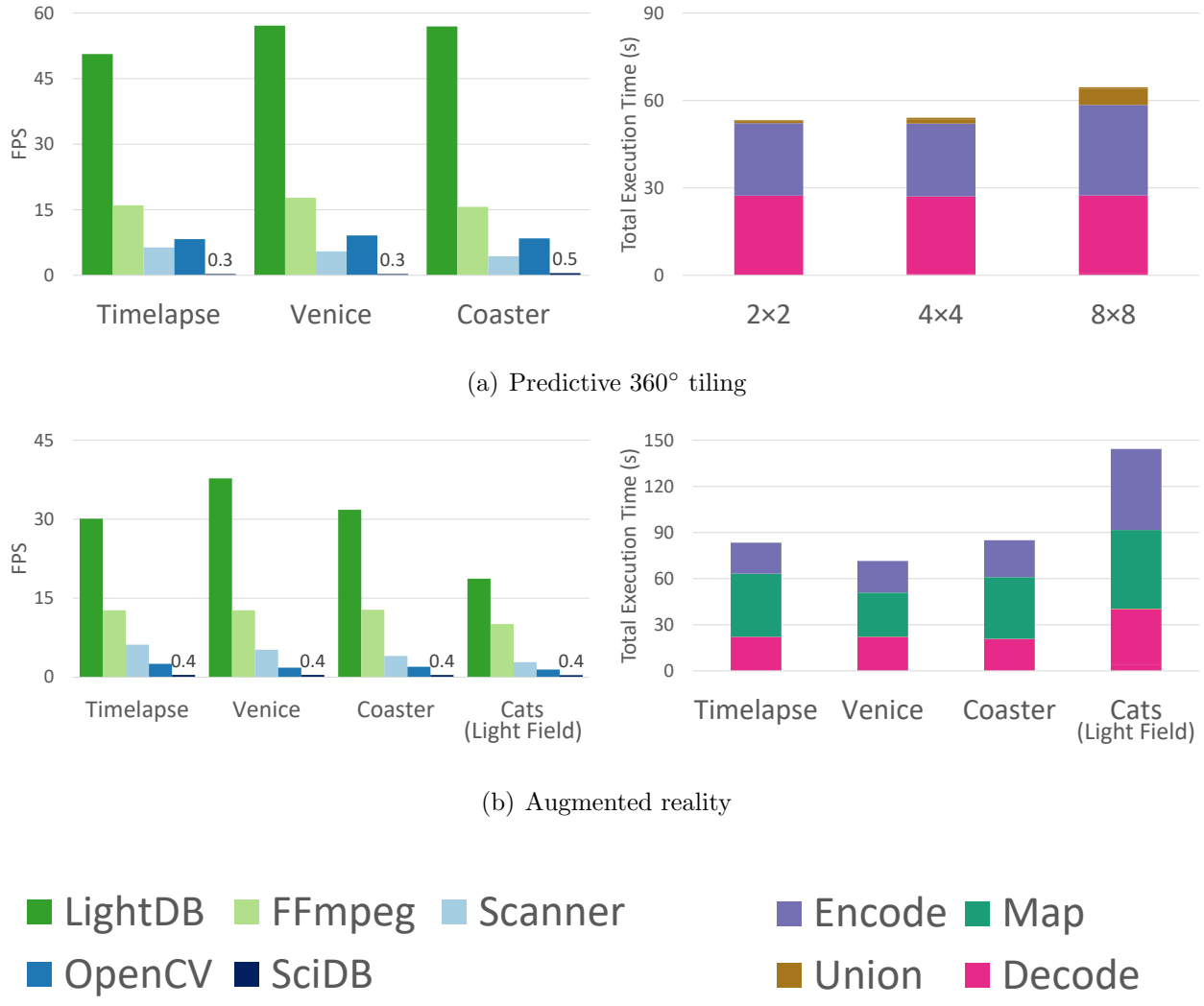
Next, we tiled the Timelapse dataset using the configuration shown in Figure 2.3. Figure 3.12(b) shows resulting tile index performance for two choices of ϕ. In this experiment, LightDB's use of the tile index is able to improve performance by allowing LightDB to decode only the relevant tile rather than the entire encoded video.

To evaluate spatial indexes, we first created a large 360TLF that simulates many 360° videos taken over time at a popular tourist destination. To do so, we repeatedly unioned the Timelapse dataset until it contained five million 360° videos defined at random points in a unit cube and at the origin. We then performed two selections with and without a spatial index defined on (x, y, z). The results are shown in Figure 3.12(c), and illustrate that the R-tree yields a modest benefit relative exhaustive search for relevant videos.

3.6 Summary

In this chapter, we presented LightDB, a video database management system (VDBMS) designed to efficiently process virtual reality (VR) and augmented reality (AR) video. LightDB exposes a data model that treats VR and AR video as a logically continuous six-dimensional light field. It offers a query language and algebra, allowing for efficient declarative queries.

We implemented a prototype of LightDB and evaluated it using several real-world applications. Our experiments show that queries in LightDB are easily expressible and yield up to a 500× performance improvement relative to other video processing frameworks.



(a) Predictive 360° tiling

(b) Augmented reality

Figure 3.7: Performance of predictive 360° and AR applications. The y-axes on the left show frames per second (FPS), while the right plot shows LightDB operator contribution to total execution time. Value for SciDB added on top of its bar.

System	% Reduced		
	Coaster	Venice	Timelapse
LightDB	78%	78%	67%
FFmpeg	75	76	71
Scanner	20	23	38
OpenCV	19	21	34
SciDB	19	23	33

Table 3.3: Percent reduction for the predictive 360° query.

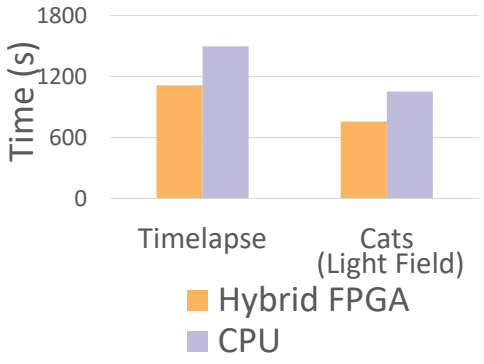
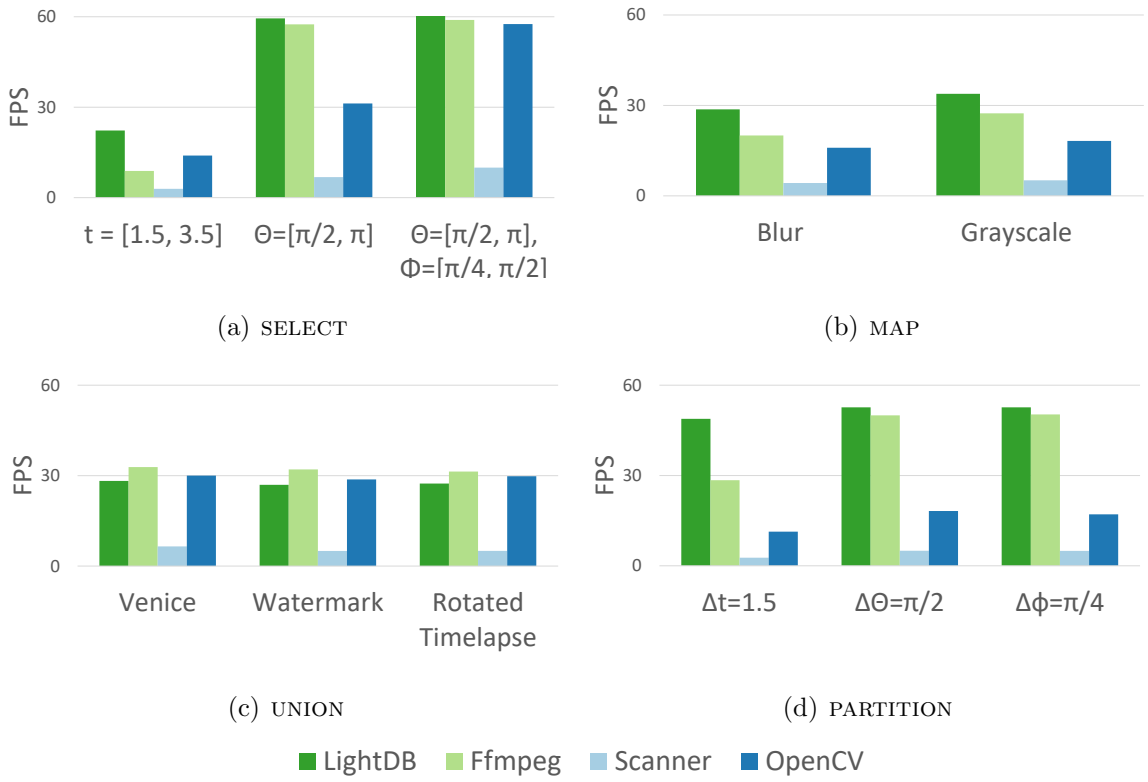


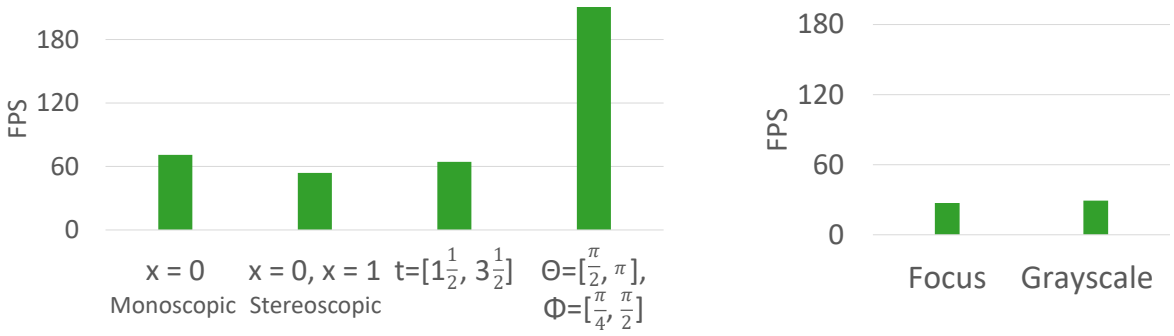
Figure 3.8: Performance of depth map generation application



(a) SELECT (b) MAP

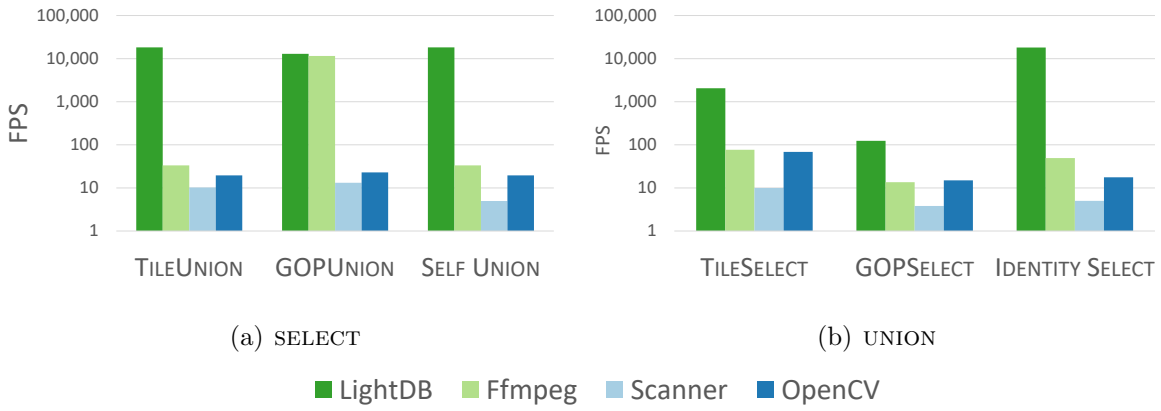
(c) UNION (d) PARTITION

Figure 3.9: PointTLF operator performance over the Timelapse dataset.



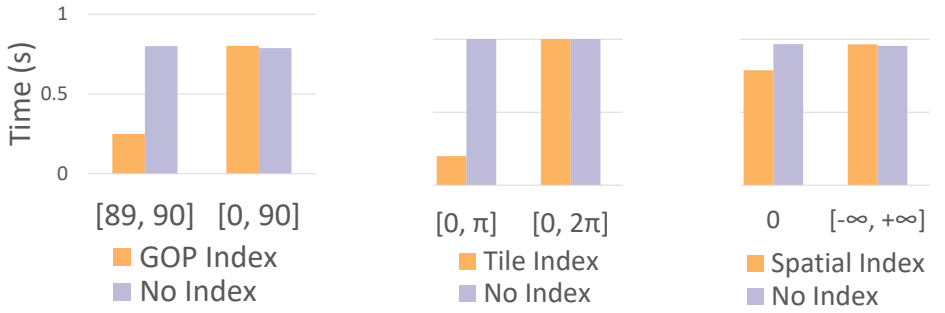
(a) SELECT (b) MAP

Figure 3.10: LightDB PlaneTLF performance over the Cats dataset.



(a) SELECT (b) UNION

Figure 3.11: Homomorphic & optimized LightDB operator performance (log scale)



(a) GOP Index (b) Tile Index (c) Spatial Index

Figure 3.12: LightDB index performance

Chapter 4

VFS: A FILE SYSTEM FOR VIDEO ANALYTICS

As we discussed in Chapter 1, the volume of video data captured and processed is rapidly increasing. YouTube receives more than 400 hours of uploaded video per minute [161]. More than six million closed-circuit television cameras populate the United Kingdom, collectively amassing an estimated 7.5 petabytes of video data per day [29]. In the United States, law-enforcement use of body-worn cameras is expected to exceed 200,000 units in service by the end of 2019 [77], collectively generating almost a terabyte of video data per day [165]. A single autonomous vehicle can generate more than 19 terabytes of video data per *hour* [69]. Recent VR light field cameras (see Section 2.3.2), which sample every visible light ray occurring within some volume of space, can produce up to a half terabyte of video data per *second*.

As we introduced in Chapter 1, a large number of systems have emerged to ingest, transform, and reason about video data to mitigate this data deluge. Each of these systems, however, suffers from the storage-related deficiencies described in Definition 1.1. Specifically, each stores data on disk as large, opaque, and independent blobs. As a result, storage-related operations on video data are inflexible and limited to reads, writes, and coarse-grained seeks. In this chapter we focus on these limited storage capabilities, which create three sets of physical challenges for application developers.

First, video-oriented applications are forced to tightly intermingle data plumbing with application logic (q.v. Definition 1.1 item 3). Developers must manually handle the (de)compression of physically-persisted video data, resolution resampling before applying machine learning and computer vision algorithms, and frame rate subsampling in network-constrained and edge-processing applications. This intermixing leads to applications that are brittle and difficult to evolve.

Second, the close coupling of application logic with physical video data storage makes it difficult to deploy optimizations and other techniques to improve application performance (q.v. Definition 1.1 item 4). For example, applications that select or produce multiple versions of a video (e.g., at different resolutions or by selecting multiple regions of interest) are responsible for (re)compressing, persisting, and selecting from amongst the potentially many versions when performing subsequent operations. Many applications avoid this additional complexity by inefficiently reapplying operations to the original video data.

Finally, many video applications collect large amounts of video data with overlapping fields of view and physically proximate locations (q.v. Definition 1.1 item 2). For example, traffic camera and surveillance networks often have multiple cameras oriented toward the same intersection. Similarly, autonomous driving and drone applications come with overlapping multiple sensors that capture nearby video data. Reducing the redundancies that occur among these sets of physically-proximate or otherwise similar video streams is neglected in all modern video-oriented systems. Leveraging spatial overlap between videos could improve performance, but doing so is difficult given that developers need to devote considerable effort dealing with low-level video compression idiosyncrasies.

In this chapter, we introduce a new *video file system (VFS)*, which is designed to decouple video application design from video data's physical layout and compression optimizations. This decoupling allows application and system developers to focus on their relevant functionality, while VFS handles the low-level details associated with video data persistence.

Analogous to relational database management systems, developers using VFS treat each video as a *logical video* and let VFS determine the best way to perform operations over one or more of the *physical videos* that it maintains on disk. To enable this independence, VFS exposes a simple and familiar file-system interface where users read and write video data through POSIX file system operations or by using VFS's command line or C++ API. Users initially write video data in any format, encoding, and resolution—either compressed or uncompressed—and VFS manages the underlying compression, serialization, and physical

layout on disk. When users subsequently read video—once again in any configuration and by optionally specifying regions of interest and other selection criteria—VFS automatically identifies and leverages the most efficient methods to retrieve and return the requested data.

VFS's interface frees users from worrying about video formats, compression, sampling, and other low-level persistence details. A user writes data in whatever video format is on hand, and later reads video data in whatever format is needed, *even if she has not previously written video in that format to VFS*.

VFS may be used standalone or integrated at the lowest level of a video database management system (VDBMS). As a standalone system, VFS improves flexibility and performance for developers and data scientists who read and write video data from and to the file system using frameworks such as OpenCV [118]. When used as a foundation for a VDBMS, the VDBMS may leverage the optimizations offered by VFS to improve query performance, especially when videos are queried multiple times.

In either mode, VFS's API supports reads and writes of video data with optional spatial (e.g., resolution, region of interest), temporal (e.g., frame rate, start and end time), or physical (e.g., compression method) predicates. By contrast, existing frameworks and VDBMSs are naively limited to monolithic or coarse-grained reads and writes of video data with fixed spatiotemporal and physical characteristics. This naive approach introduces substantial overheads that VFS is able to avoid using the optimizations we describe below.

Under the hood, VFS 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, VFS decomposes each video into sequences of contiguous, independently-decodable, optionally-compressed sets of frames. VFS then indexes those video fragments and the resulting structure allows VFS to read only the minimal set necessary to satisfy a read operation. As VFS 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. Later, when a user performs a subsequent read operation, VFS leverages a minimal-cost

```
# Write initial video
$ vfs write traffic.mp4
# Read small RGB version
$ vfs read traffic.rgb -r 320x180
# Read 1K RGB between time 5 and 20
$ vfs read traffic.rgb -r 1280x720 -s 5 -e 20
# Read 1K RGB cropped at 640x720 between time 5 and 20
$ vfs read traffic.rgb -r 1280x720 -s 5 -e 20 -w 640 -h 720
# Read the same video as above using the POSIX interface
$ cat /vfs/traffic/1280x720s5e20x640y720.rgb > traffic.rgb
```

Figure 4.1: Example commands using the VFS command-line interface to read and write traffic video data.

subset of these views to generate its answer. Finally, because video fragments can arbitrarily overlap and have complex interdependencies, VFS uses a satisfiability modulo theories (SMT) solver to identify the best sets of views to satisfy a request.

Second, to trade off read performance with the storage size of the materialized view collection, VFS exposes an application-specified video storage budget. This budget allows administrators to balance between these factors. When a video's storage budget is exceeded, VFS prunes stale views by selecting those least likely to be useful in answering subsequent queries and, among equivalent entries, VFS optimizes for quality and defragmentation.

Finally, VFS 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 the overlapping region only once. Developers may explicitly indicate videos eligible for joint compression or may allow VFS to automatically identify eligible video data.

Collectively, by decoupling video-oriented systems and applications from the underlying physical representation of video data, VFS allows for easier application development, faster read and write performance for computer vision and machine learning pipelines, and lower storage costs for physically proximate video datasets.

The remainder of this chapter is organized as follows. First, in Section 4.1, we give a high-level description of the architecture and API of VFS. We then detail the three major functions of VFS. First, VFS must choose the most efficient way to answer a read query, and we describe the query answering approach used by VFS in Section 4.2. Second, when ingesting video data, VFS must efficiently physically arrange and persist the written video data on disk. We describe the approach used by VFS in Section 4.3. Next, in Section 4.4 we discuss data compression techniques automatically deployed by VFS for serialized video data. We finally evaluate the performance of VFS in Section 4.5.

4.1 Architecture

Consider an application designed to monitor an intersection for license plates associated with missing children or adults with dementia. A typical implementation of such an application 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 license plates. Such models are typically deep learning models that are expensive to execute. Therefore, the application might first use an inexpensive low-resolution video representation to prune frames that are unlikely to contain license plates at all. VStore [164], for example, implements this type of optimization, but requires the application specify beforehand that it needs both a low- and high-resolution approximations. Other image processing systems use similar optimizations [85, 99]. Once vehicles with potentially matching plates are found in the video, the application might apply another machine learning algorithm to cross reference the vehicle make and model. This third processing step may require the video in yet another representation. Finally, a user might request and view all

video sequences containing likely candidates. This might involve further converting to a representation compatible with the viewer (e.g., at a resolution compatible with a mobile device or compressed using a supported codec).

An application working with video files stored in a regular file system must manually convert to and from these various representations, while an application that utilizes a system like VStore must decide a priori every possible format or resolution that might be needed in the future. On the other hand, an application leveraging VFS can simply read video data in the desired form. For example, when the above application wants low-resolution, uncompressed video data to find frames with license plates, it can directly read the file `traffic/320x180.rgb` from VFS (or execute the command line variant `vfs read traffic.rgb -r 320x180`), letting VFS be responsible for efficiently producing the desired data.

Figure 4.1 shows additional examples of VFS commands using the VFS command-line interface. A `vfs write` command ingests video data into VFS, which creates a logical entry for the video (if it does not already exist) and writes the video data as an initial physical video. The `vfs read` command produces a new file on disk containing the requested video fragment at the desired resolution, times, and physical format (if it has not already been cached).

Critically, VFS automatically selects the most efficient way to generate the desired video data in the requested format 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 region, e.g., `traffic/1280x720x640y360.rgb`, requests a 1280×720 resolution video cropped to the region defined by the rectangle at $(0, 0)$ and $(640, 360)$.

Table 4.1 summarizes the full set of operations that VFS exposes. Importantly, these operations are over *logical videos*, which VFS executes to produce or store *physical video* data. Each operation involves a point- or range-based scan or insertion over a single logical video source. VFS allows constraints on any combination of temporal (T), spatial (S), and physical (P) parameters. Temporal parameters include start and end time interval ($[s, e]$) and frame

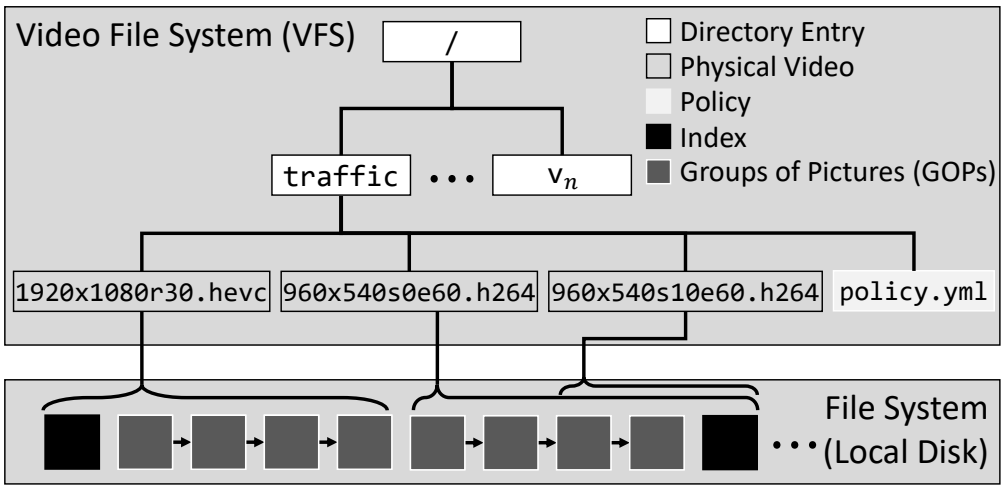


Figure 4.2: The video file system (VFS) directory structure. Each video is organized under its own directory entry, which contains physical videos and a policy file. Each physical video is associated with a sequence of optionally-compressed video segments called groups of pictures (GOPs).

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) and compression method (c; e.g., HEVC).

Besides the POSIX-compliant file system interface (Table 4.1), VFS also provide a C++, Python, and OpenCV-compatible API, and a command-line interface similar to that exposed by HDFS [140] (Figure 4.1). Our prototype treats the POSIX interface as primary, and each of the other APIs delegate to it.

An exemplary directory hierarchy of VFS is illustrated in Figure 4.2. To create the missing-person application described above, a user simply executes `vfs write` or writes each traffic camera's video using the POSIX interface (e.g., to `/vfs/traffic.mp4`). In both cases, VFS implicitly executes *create*(`traffic`), which creates a new logical video and directory, `traffic`, on the file system. It then automatically extracts the video's metadata for spatial, temporal, and physical parameters, and uses these to execute *write*(`traffic`, S, T, P). For example,

Table 4.1: VFS operations. Both reads and writes require specification of spatial (S; resolution, region of interest), temporal (T; start and end time, frame rate), and physical (P; frame layout, compression codec) parameters.

Op	Parameters
read	$(name, S, T, P)$
write	$(name, S, T, P, data)$
create	$(name)$
delete	$(name)$
policy	$(name, parameter, value)$
pin	$(name, S, T, P)$
unpin	$(name, S, T, P)$

if the video `traffic.mp4` contained video data at 2K resolution, 30 frames per second, and YUV420 format, the `vfs write` would create a new file `traffic/1920x1080r30fyuv420.hevc`. In cases where VFS cannot automatically extract video parameters (e.g., when writing raw uncompressed data), the user must specify them explicitly.

Later, the application may read this or other representations of the logical video from VFS. Critically, rather than being able to read only the previously-written variant, users may read the logical video using *any* combination of valid parameters and are not limited to reading previously-written results (e.g., opening and reading `/traffic/999x999r99.h264` would constitute a valid read). Regardless of what an application reads, VFS transparently manages decompression and resampling, freeing the application from needing to be involved.

Under the hood, VFS arranges each 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 contain raw pixel data or be compressed using a video codec. Compressed GOPs, however, are constrained such that they are independently decodable and may not take data dependencies on other GOPs.

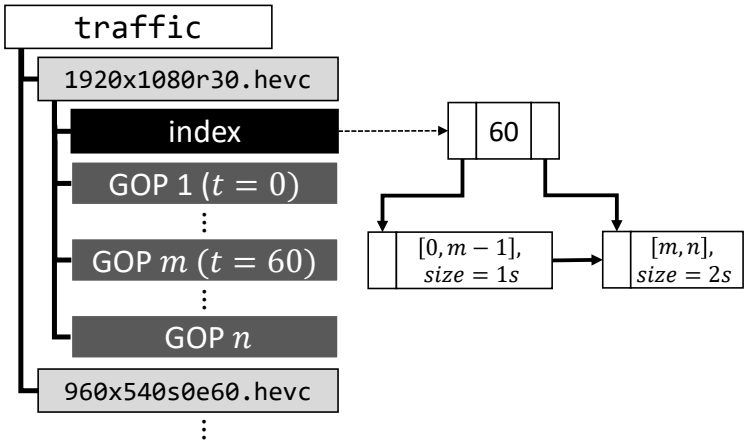


Figure 4.3: An example VFS physical organization that contains one logical video and two underlying physical videos. For physical video `1920x1080r30.hevc`, the first m GOPs are each one second in length, while the remaining $n - m$ are two seconds. These durations are recorded in the associated index. Shading is as shown in Figure 4.2.

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 VFS accepts as-is ingested compressed GOP sizes (which are typically less than 512kB). However, partitioning extremely long GOPs on ingest is a straightforward enhancement. For uncompressed GOPs, our prototype implementation automatically partitions video data into blocks of size $\leq$ 25MB (the size of a single RGB 4K frame), or a single frame for resolutions that exceed this threshold.

Figure 4.3 illustrates the internal physical state of VFS after executing the write described above. VFS has created a directory that contains the previously-written physical representation for the logical video `traffic`. VFS has stored each GOP in this representation on disk as a series of distinct files `v/1920x1080r30.hevc/1`, ..., `v/1920x1080r30.hevc/n`. It has also constructed a non-clustered temporal index that maps from time to the GOP file containing visual information for that time. This level of detail is invisible to applications, which access VFS only through the operations summarized in Table 4.1 and also illustrated in Figure 4.1.

Table 4.2: Video policy parameters and default values.

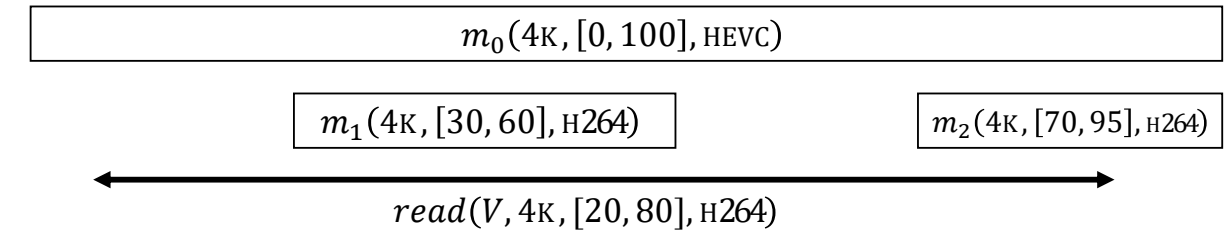
Parameter	Default	Description
Quality threshold	35	Maximum quality degradation (PSNR)
Storage budget	10×	Per-video cache size
Joint compression	∅	Joint compression candidates

Finally, each logical video is also associated with a *policy* that determines the hyperparameters (listed in Table 4.2) used to tune read and write performance. Policies are modified using the *policy* API method shown in Table 4.1. Developers may also *pin* (and *unpin*) regions of a logical video to prevent their being evicted from VFS's internal cache. Pinned regions are not counted against the video's storage budget.

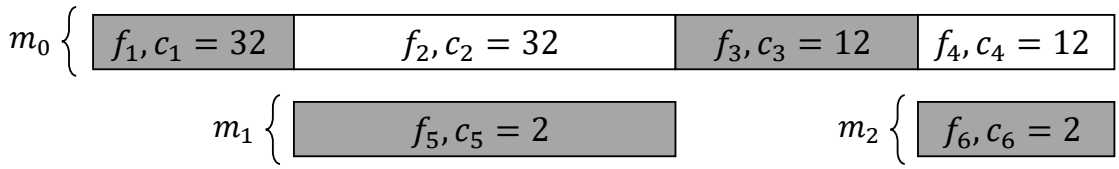
4.2 Data Retrieval

As mentioned, VFS internally represents each logical video as a collection of one or more cached physical videos. When executing a read, VFS must produce the result using one or more of these physical videos.

Consider a simplified version of the alert application described in Section 4.1, where a single camera has captured 100 minutes of HEVC-encoded video and written it to VFS 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, VFS now contains the original video (m_0) and the cached versions of the two fragments (m_1, m_2) as illustrated in Figure 4.4(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 format (HEVC or H264).



(a) Read operation on three cached physical videos



(b) Physical video fragments with simplified cost formulae

Figure 4.4: Figure 4.4(a) shows the query $read(V, 4K, [20, 80], H264)$ executed over a video with cached physical videos $\{m_0, m_1, m_2\}$. Figure 4.4(b) shows the weighted physical video fragments using simplified cost formulae. The lowest-cost result is shaded.

Later, a first responder on the scene views a one-hour portion of the recorded video on her mobile device, which only has hardware support for H264 decompression. To deliver this video, the application executes $read(V, 4K, [20, 80], H264)$, which, as shown at the bottom of Figure 4.4(a), requests video V between time $[20, 80]$ in spatial configuration 4K and physical configuration H264.

VFS responds by first identifying subsets of the available physical videos that can be leveraged to produce the result. For example, VFS 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 VFS need not incur a heavy transcoding cost for these regions. Figure 4.4(b) shows the different selections that VFS might make to answer this read. Each *physical video fragment* $\{f_1, ..f_6\}$ in Figure 4.4(b) represents a

different region that VFS might select. Note that VFS need not consider other subdivisions of these fragments—for example by subdividing f_5 at time $[30, 40]$ and $[40, 60]$—since f_5 being cheaper at $[30, 40]$ implies that it is at $[40, 60]$ too.

To model these transcoding costs, VFS 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.

VFS must also ensure that the quality of a result has sufficient fidelity. For example, using a heavily downsampled physical video (e.g., 32×32 pixels) to answer a read requesting 4K video is likely to be unsatisfactory. To avoid this, VFS adopts 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, VFS 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 Table 4.1); otherwise a default threshold is used ($\epsilon = 35$ 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 [71]. Values $\geq$40 are considered to be lossless qualities, while $\geq$30 are near-lossless.

Given this cost and quality model, we now turn to how VFS selects fragments for use in performing a read operation. In general, given a *read* operation and a set of physical videos, producing a result requires VFS to perform several operations. First, it must select fragments that cover the desired spatial and temporal ranges. To ensure that a solution exists, VFS maintains the initially-written video (the *root physical video* m_0), and VFS returns an error for reads extending outside of the temporal interval of m_0.

Second, when the selected physical videos temporally overlap, VFS 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 reminiscent of prior work in materialized view selection [60]. Fortunately, a VFS 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 4.4(a). First, observe that the collective start and end points of the physical videos form a set of *transition points* where VFS may opt to switch to an alternate physical video.

In Figure 4.4(a), the transition times include those in the set $\{30, 60, 70\}$, and we illustrate them in Figure 4.4(b) by partitioning the set of cached physical videos 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.

Now observe that between each consecutive pair of transition points VFS must choose exactly one physical video fragment. In Figure 4.4(b), we highlight one such set of choices that covers the read interval. Each choice of a fragment comes with a cost (i.e., f_1 has cost 32), derived using a cost formula that assigns a transcode cost equal to (i) twice the fragment's duration if it requires decoding and then re-encoding, (ii) the fragment's duration if it only requires encoding, and (iii) zero for fragments already in the target physical format P_q (e.g., H264 for the example in Figure 4.4). Formally:

$$c_t(f) = \begin{cases} 0 & \text{if } \textsc{physical-format}(f) = P_q \\ \textsc{duration}(f) & \text{if } \textsc{uncompressed}(f) \\ 2 \cdot \textsc{duration}(f) & \text{otherwise} \end{cases} \tag{4.1}$$

To conclude our example, observe that we must choose a set of physical video fragments that (i) cover the queried temporal range, and (ii) do not temporally overlap. Further, of all the possible paths between 20 and 80, the one with the lowest cost—highlighted in Figure 4.4(b)—minimizes the total cost of producing the answer. These characteristics collectively meet the requirements identified at the beginning of this section.

4.2.1 Interval Cover Fragment Selection

We propose two algorithms to solve the cost minimization problem. The first targets videos without *dependent frames*, while the second (to be discussed in Section 4.2.2) considers frames of this type.

Our first algorithm models the problem as a weighted interval cover problem, with each fragment being temporally arranged on a real-number line. Weights for each fragment are given by the cost function c, and only fragments admitted by the quality model u are considered. Given a VFS read, we identify the minimum-cost set of physical view fragments as follows:

Problem (Cached physical video fragment interval cover). Consider a $read(D, S_q, T_q, P_q)$ operation over logical video D, which is associated with cached physical videos $M = \{m_1, ..., m_n\}$ each having configuration (S_i, T_i, P_i). We are given a function $C(m, T)$ that converts physical video m into configuration (S_q, T, P_q) at cost $c(m, T)$, and a function $\oplus$ that concatenates physical videos with identical configurations. We want to find a result $A = C(m_0^*, T_0^*) \oplus \cdots \oplus C(m_k^*, T_k^*)$ that is:

- **Valid** such that it is *non-overlapping* (i.e., $T_i^* \cap T_j^* = \emptyset$) and is a *cover* of T (i.e., $\bigcup T_i^* = T_q$).
- **Minimal cost** such that $\forall\{(m, T) \mid m \in M, T \subseteq T_q\}\colon \sum c(m_i, T_i) \geq c(A)$.

Definitions. Let (s_i, e_i) be the start and end time of physical video m_i and (s_q, e_q) be the start and end times of the read operation (i.e., T_q). Let $W = \{s_q, e_q, s_1, e_1, ..., s_n, e_n\}$ be the set of all such start and end times. Finally, let $R_i = \big\{p \mid p \in W \times W, \max(s_i, s_q) \leq p \leq \min(e_i, e_q)\big\}$ be the transition points associated with m_i.

Solution. We begin by identifying the physical video fragments relevant to a solution (i.e., $\{f_1, ..., f_6\}$ in Figure 4.4(b)). The start and end times associated with each such fragment are defined by the intervals $F_i = \big\{[\alpha, \beta] \mid \alpha, \beta \in R_i, \alpha < \beta, \nexists\gamma \in R_i\colon \alpha < \gamma < \beta\big\}$. The size $|F_i|$ is at most $\mathcal{O}(|M|)$, the number of physical videos. Collectively, let $F = \bigcup F_i$ be the set of all fragments, and because $|F_i| = \mathcal{O}(|M|)$, $|F| = \mathcal{O}(|M|^2)$. We associate weight $c(f)$ to each fragment $f \in F$.

Using the fragments, which cover the interval (s_q, e_q), we wish to find a minimum-weight cover. While covering problems are NP-hard in most contexts, interval covering problems are solvable in polynomial time [126]. As illustrated in Figure 4.4, our formulation of

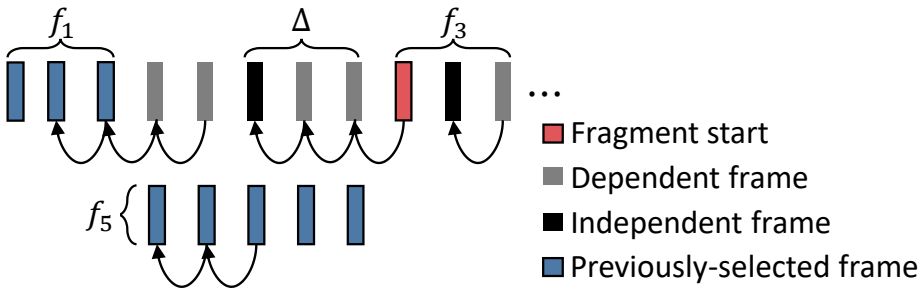


Figure 4.5: In this simplified illustration based on the fragments $\{f_1, f_3, f_5\}$ in Figure 4.4, VFS is considering using fragment f_3 starting with the frame highlighted in red, and has already decided to use f_1 and f_5. However, this frame cannot be decoded without also transitively decoding the dependencies represented by directed edges and labeled Δ. VFS look-back cost c_l is a function of these frames, with the single independent frame being more expensive than the two dependent frames.

fragment selection involves selecting minimum-cost intervals on the (temporal) positive real number line. Since no intervals partially overlap, we sort the intervals by start time and sequentially select the fragment with lowest cost. This approach is a special case of the greedy weighted interval cover algorithm, which generates a minimum-weight cover R on the interval (s_q, e_q) [12]. Applying this algorithm has complexity linear in the number of intervals, or $\mathcal{O}(|F|) = \mathcal{O}(|M|^2)$.

4.2.2 Constraint Satisfaction Selection

The previous solution is inexpensive and optimal for selecting subsets of videos that have transcode cost proportional to duration or number of frames. However, as video compression codecs utilize data dependencies between frames, this assumption is frequently violated. We model these data dependencies as a directed graph, with frames as vertices and directed edges representing data dependencies between pairs of frames. For a given fragment, we denote Δ as the set of reachable frames not part of the fragment itself (i.e., its *dependent frames*).

As a concrete example, consider Figure 4.5, which shows the frames within a physical video with their data dependencies indicated by arrows. If VFS wishes to use fragment f_3 starting at the red-highlighted frame, it must first decode all of the red frame's dependent frames, labeled Δ in Figure 4.5. This violates the assumption in the transcode cost model above: the cost of transcoding frame depends on where within the video it occurs, and whether its dependent frames are also transcoded.

To model this cost, we add a *look-back cost* $c_l(P, f)$ that gives the cost of decoding the frames in fragment f along with the dependent frames Δ 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 4.5, these dependencies come in two forms: independent frames $\mathrm{A} \subseteq \Delta$ (i.e., frames with out-degree zero in our graphical representation) which are more expensive to transcode, and dependent frames (those with outgoing edges) which are inexpensive but require first decoding dependencies. On our hardware, we empirically found independent frames to be approximately five times more expensive to decode than dependent frames, and so our prototype sets $\eta = 5$. Formally, we define c_l as:

$$c_l(P, f) = \eta \cdot |\mathrm{A} - P| + |\Delta - \mathrm{A} - P| \tag{4.2}$$

Optimizing the VFS *robust cost model* $c_r(P, f) = c_l(P, f) + c_t(f)$ requires jointly optimizing both c_l (see Equation 4.2) and c_t (see Equation 4.1), where each fragment choice affects the transitive dependencies P of future choices. This problem is not solvable in polynomial time, so we employed a SMT solver [37] in order to generate an optimal solution to c_r. Our embedding constrains frames in overlapping fragments so that only one is chosen, 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. Solving this is typically more expensive than the interval cover problem, and we evaluate our two algorithms in Section 4.5.1.

4.3 Data Caching

In the previous sections we described how VFS persists, reads, and writes physical videos. We now describe how VFS 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 operation, should VFS 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 VFS discard?

To aid in both of these decisions, VFS maintains a video-specific *storage budget* that limits the total size of the physical videos associated with each logical video. The storage budget is stored in each video's policy (see Table 4.2) 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 root physical video in our prototype). VFS never evicts the root physical video (see Section 4.2), which is always available in the cache. Pinned regions of a logical video (see Section 4.1) are not counted against a logical video's storage budget, and VFS does not consider pinned regions for cache eviction.

As a running example, consider the sequence of reads illustrated in Figure 4.6, which mirrors the alert application commands shown in Figure 4.1 and described in Section 4.1. In this example, an application reads a low-resolution uncompressed video from VFS for use with a license detection algorithm. VFS 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. VFS 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. VFS 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.

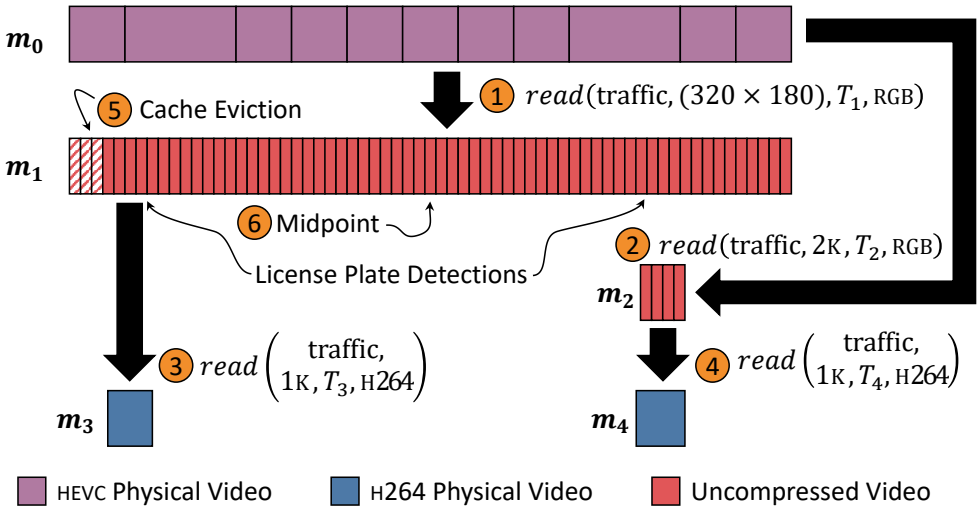


Figure 4.6: VFS caches read results and uses them to answer future queries. In ❶ an application reads logical video `traffic` at 320×180 resolution for use in license plate detection (see command two in Figure 4.1) and VFS caches the result as m_1. In ❷ VFS caches m_2, a region with a dubious license plate detection (command 3 in Figure 4.1). In ❸ and ❹ VFS caches H264-encoded m_3 & m_4, where license plates were detected. However, reading m_4 exceeds the storage budget and VFS responds by evicting the striped region at ❺. Finally, ❻ shows the midpoint of m_1 which is used in calculating the modified LRU score for eviction.

The key idea behind VFS's cache is to logically break physical videos into "pages." That is, rather than treating each physical video as a monolithic cache entry, VFS targets the individual GOPs *within* each physical video. Using GOPs as cache pages greatly homogenizes the sizes of the entries that VFS must consider. VFS'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 [18]).

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

- One GOP might be a higher-resolution 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 quadratic complexity in the number of fragments; see Section 4.2).

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, our eviction policy modifies LRU in the following ways:

- We decrease the recency of GOPs occurring at the beginning and end of a physical video more highly than those in the middle.
- We decrease the recency of the lowest-quality (as given by our quality cost model; see Section 4.2) GOP that has higher-quality redundant copies.

Formally, let $r(g)$ be the normalized (i.e., range$(r) = [0, 1]$) distance of a GOP g from the midpoint of a sequence of contiguous physical video entries that includes g (as an example, Figure 4.6 highlights the midpoint of video m_1). Let $q(g) = \min(u(g_0, g), 40)$ be the quality of g compared to its associated root physical video GOP clamped at 40, which we threshold as lossless quality (see Section 4.2 for a definition of u). Our modified policy $MLRU$ is then defined by $MLRU(g) = LRU(g) - \alpha \cdot r(g) - \beta \cdot q(g)$. Through empirical observation our prototype implementation sets $\alpha = 10$ and $\beta = -\frac{1}{40}$.

In Figure 4.6, we show VFS choosing to evict the three-frame GOP at the beginning of m_1 and to cache m_4. If our prototype had weighed the second modification more heavily than the first, VFS would instead elect to evict m_3, since it was not recently used and is the variant with lowest quality.

4.4 Data Compression

To improve the storage performance of written and cached video data, VFS employs two compression-oriented optimizations and one optimization that reduces the number of physical video fragments, each of which we describe in this section. Specifically, VFS (i) jointly compresses redundant data across multiple physical videos (Section 4.4.1); (ii) lazily performs compression on blocks of uncompressed, infrequently-accessed GOPs (Section 4.4.2); and (iii) improves the performance of reads by compacting temporally-adjacent physical videos (Section 4.4.3).

4.4.1 Joint Physical Video Compression

Many video applications capture video 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. 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.

VFS optimizes storage of these videos by reducing the redundancy between pairs of highly-similar video streams. This optimization, which we call *joint compression*, is applied between pairs of *logical videos* written to VFS. VFS's joint compression optimization currently assumes stationary or slowly-translating cameras; the modifications required to support dynamic and translating cameras are highlighted below and left as future work.

In applying the optimization, VFS examines cached physical videos across pairs of logical videos. For each pair of frames found, it identifies a region of overlap between the frames and combines them. This process is transparent to users, who can continue to logically read the individual videos as if they were stored separately on disk.

Figure 4.7 illustrates this process on two frames taken from a synthetic dataset (Visual Road-1K-50%, described in Section 4.5). Figure 4.7(a) and Figure 4.7(b) respectively show the two frames with the overlapping regions highlighted, and Figure 4.7(c) shows the overlapping regions combined.

Critically, because the frames were captured from cameras at different orientations, combining them requires more than an isomorphic translation or rotation (e.g., the angle of the horizontal sidewalk is not aligned in the two frames). Instead, a homography between the two frames is estimated and the result used to transform between the two spaces. As shown in Figure 4.7(c), VFS 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.

More specifically, VFS applies joint projection by executing the following steps. First, it estimates a homography between the two frames. To do so, it waits to receive two frames from the pair of videos being jointly compressed (or reads two frames from video pairs being lazily compressed). Next, it applies a feature detection algorithm (SIFT [97]) that identifies similar features that co-occur in both frames. Using these features and random sample consensus (RANSAC [50]), it estimates the homography matrix used to transform between frame spaces. Our current prototype applies SIFT and RANSAC once per second of video (generally every thirty frames), which suffices for stationary cameras. For dynamic cameras, however, it would be advantageous to adaptively adjust the recalculation interval to avoid a loss in quality, and we leave this as future work.

With the homography estimated, VFS uses it to transform 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. VFS splits these into three distinct regions and uses an ordinary video codec to encode each region separately.

This process is formalized in Algorithm 4.1. The function JOINT-COMPRESS iterates over each frame in videos F and G and estimates homography every thirty frames. It then uses the JOINT-COMPRESS-FRAMES to perform pairwise joint compression on each frame f and g by computing x_f, the right extent of the overlap in the left frame and x_g, the left extent of the overlap in the right frame. It uses these cut-points to subdivide f and g into left, overlapping, and right regions and compresses each individually.

This joint compression optimization process is activated in one of two ways. First, a user may explicitly request joint compression between two videos by modifying a pair of videos' policies (see Table 4.2). Even when specified in the policy, VFS will only apply joint compression when the estimated homography indicates that the amount of overlap exceeds an administrator-specified threshold (25% in our prototype).

Alternatively, VFS lazily examines pairs of physical videos and applies joint compression by randomly sampling GOPs of candidate physical videos and evaluating them for overlap. To prevent false positives, when lazily compressing physical videos VFS applies joint compression only for pairs of physical videos that have substantial overlap with high RANSAC confidence, as specified as an administrator-configurable threshold.

4.4.2 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 4.5 is 5.2TB uncompressed as 8-bit RGB). As VFS caches uncompressed video regions as the result of reads (e.g., while scanning for license plates), it quickly accumulates

many uncompressed cache entries that risk exhausting the video's storage budget. In this section we mitigate this risk by compressing these entries and balancing compression speed with cache entry size.

To avoid this issue, one option is to perform lossless compression on uncompressed frames prior to caching them. However, doing so when executing the read increases the operation latency and leads to decreased performance for applications that request only small amounts of high-resolution uncompressed frames. An alternative approach would be to begin performing lossless compression during reads when the associated video's storage budget exceeds some threshold. For example, VFS might allocate half of its budget for uncompressed data and compress cached entries after this amount is exceeded. This would increase read latency in exchange for improved cache performance.

This approach, however, suffers from two disadvantages. First, it never prompts the system to compress the data that was cached prior to the budget threshold being exceeded. Second, the new read may produce a cache entry that is less useful for future reads than an alternative cache entry (e.g., it might be immediately evicted). This means that the computational expense of compression is wasted.

To avoid these disadvantages, VFS adopts the following approach. When a video's cache size exceeds some threshold (25% in our prototype), VFS activates a special *deferred compression* mode. In this mode, when a read requests uncompressed data, VFS examines the current cache and orders the uncompressed physical video entries by eviction order. It then performs lossless compression on 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 [46].

VFS 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). VFS linearly scales Zstandard compression level with remaining storage budget, which has the effect of decreasing compressed size while increasing compression time. Second, while deferred compression is active, VFS continues to compress cache entries in a background thread during periods when no other IO requests are being executed.

4.4.3 Physical Video Compaction

As a result of caching the result of user queries, VFS 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 VFS 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, VFS periodically compacts pairs of contiguous cached videos and substitutes a unified representation. To do so non-quiescently, it applies the algorithm shown in Algorithm 4.2. This algorithm 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.

4.4.4 Summary of Data Compression Optimizations

Collectively through the data compression optimizations described in this section, VFS improves storage performance in three interrelated ways. First, it applies joint physical video compression to reduce the redundancy *between* pairs of otherwise unrelated logical videos. Second, it applies deferred compression to reduce the redundancy of physical video data *within* a single video fragment cache entry. Finally, it applies its video compaction optimization to improve the performance of *sequences of* physical video cache entries.

4.5 Evaluation

We have implemented a prototype of VFS using approximately 5,000 lines of C++ code. GPU-based operators were implemented using CUDA [114] and NVENCODE/NVDECODE [113]. We use the `libfuse` file system in userspace (FUSE) reference implementation to export VFS to the operating system kernel and expose the POSIX-compliant interface for consuming applications [92]. VFS also uses FFmpeg [17] for some video plumbing operations such as GOP segmentation and concatenation. Our prototype currently adopts a no-overwrite policy for logical videos and disallows updates. We plan on supporting both of these features in a future release. Finally, when performing writes, VFS does not guarantee that data are visible to other readers until the file being written is closed.

We evaluate VFS by comparing it to two baseline systems (and directly against the local file system) in terms of read (Section 4.5.1), write and caching (Section 4.5.2), and compression (Section 4.5.3) performance.

Baseline systems. We compare VFS against VStore [164], a recent storage system that supports video analytic workloads by pre-computing all possible video representations. We also evaluate VFS 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 all experiments on VStore are limited to this size. The local file system is formatted using `ext4` and backed by a SSD drive.

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 VFS 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 4.3 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 [102]. The dataset is provided as 7,494 separate images, which we converted into a video using H264 at 30 frames per second and one-second GOPs.

The "Waymo" dataset is an autonomous driving dataset [158]. 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 the video analytics benchmark described in Chapter 5. This benchmark, called Visual Road, is specifically designed to evaluate the performance of video-oriented data management systems [66]. 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, where each video has 75% horizontal overlap with the other.

Because the size of the uncompressed 4K Visual Road datasets exceed the storage capacity of our experimental system, we do not show results for this dataset that require fully persisting its uncompressed representation to disk.

4.5.1 Data Retrieval Performance

Read Performance. Our first experiment explores the read performance of VFS 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, we repeatedly execute queries of the form $read(\text{VisualRoad-4K-30\%}, 3840{\times}2160, [t_1, t_2], P)$, with times $0 \leq t_1 < t_1 + 1 \leq t_2 < 3600$ (in seconds) along with a physical format $P \in \{\text{H264}, \text{HEVC}, \text{RGB}, \text{YUV420}, \text{YUV422}, \text{NV12}\}$, with t_1, t_2, and P drawn uniformly at random. These cache entries might be generated by

application of a machine learning algorithm (e.g., license plate detection) over many regions of a video. We iterate this process until VFS 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 VFS to leverage its cached physical video fragments.

We show the performance of this read in Figure 4.8 for the two application scenarios. 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 4.8(a), even a cache with a small number of entries is able to improve read performance by a substantial amount—28% at 100 entries and up to a maximum improvement of 54%. We further observe that the constraint satisfaction algorithm outperforms its interval cover counterpart. This is because the solution it finds requires decoding fewer redundant dependent frames. However, while for 4K video the cost of applying both algorithms does not constitute a significant fraction of the read operation, as we show in Figure 4.8(b) the exponential cost of constraint satisfaction diminishes the performance benefit as the number of fragments grows large. Switching to interval cover for extremely large caches would likely improve performance in these edge cases, though we leave this as future work.

Read Format Flexibility. Our next experiment evaluates VFS'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 VFS, VStore, and the local disk. We write the data to each file system in both compressed (224MB) and uncompressed form (approximately 328GB).

We use an empty cache for VFS and read the persisted videos from each system in various formats and measure the throughput offered by each system. Figure 4.9 shows results for a read in the same format (Figure 4.9(a)) written to a file system and different formats (Figure 4.9(b)). Because the local file system and 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.

We find that read performance *without* a format conversion from VFS is modestly slower than the local file system, due in part to recently-identified bottlenecks in FUSE [151], the local file system being able to execute entirely without kernel transitions, and the need for VFS to concatenate many individual GOPs. However, our results show that VFS is able to adapt to reads in *any* format, a benefit not available when using the local file system.

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

Fragment Selection Performance. Our final read experiment evaluates the overhead associated with the interval cover and constraint satisfaction fragment selection algorithms using 30-frame GOPs (i.e., one independent frame and 29 dependent frames). The dashed lines in Figure 4.8(b) shows the time required to execute each of the algorithms in isolation, without the accompanying read.

We find that the overhead of selecting physical video fragments is low relative to the cost of producing the output, and that the advantage offered by utilizing the physical video fragments outweighs this cost except for a cache with many single-frame entries.

4.5.2 Data Persistence & Caching

Write Throughput. Our next evaluation explores VFS write and caching performance. To evaluate the write performance of VFS 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 4.10(a).

For datasets that will fit on local storage, VFS performs similarly to using the local file system and VStore, though VFS 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, VFS is able to activate its deferred decompression optimization and automatically begin compressing the data being written. This compression allows it to store datasets that no other system can handle, albeit at the cost of decreased throughput.

We next write the compressed evaluation datasets to each store. Figure 4.10(b) shows the performance results for each baseline system. Here all systems perform approximately equivalently, with both VFS and VStore exhibiting minor overhead relative to the local file system.

Cache Performance. To evaluate the VFS cache eviction policy, we repeat our experimental setup for read performance in Section 4.5.1. We execute 5,000 random read operations to populate the VFS 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 VFS eviction policy. This has the effect of limiting 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 4.11 shows the runtimes for each policy and storage budget. These results show that (i) that VFS is able to reduce read execution by approximately 14%, even with a limited budget, relative to application of a LRU policy.

4.5.3 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 VFS. We then subsequently read each video back from VFS 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 4.4 gives the PSNR for recovered data compared against the written videos. Recall that a PSNR of $\geq$40 is considered to be lossless, and $\geq$30 near-lossless [71]. In general, we find high quality recovery for the left input to jointly compressed videos, and near-lossless degradation for the right input. This difference is due to our approach of copying the left frame onto the right and a resulting loss in fidelity when performing the inverse projection on the right frame. A potential enhancement, which we leave as future work, is to instead write the mean of the overlapping pixels or to interlace them.

Joint Compression Throughput. Our next experiment examines VFS read throughput with and without the joint compression optimization applied. For this experiment, we write each video in the VisualRoad-1K-30% dataset to VFS, 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 4.13(a) shows the throughput achieved when executing the read using each configuration. Our results indicate that read overhead for videos stored using joint compression is modest and similar to reads that are not co-compressed.

Applying joint compression to a pair of videos requires a number of nontrivial operations, and our final experiment evaluates the overhead associated with its execution. For this experiment, we write each video in each Visual Road dataset to VFS and measure throughput. Figure 4.13(b) shows the results of this experiment. Surprisingly, joint writes are *faster* than writing each video stream separately. This speedup is due to VFS'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 identifying homography (approximately 0.5 milliseconds for every GOP) and partitioning frames (approximately 2 milliseconds applied per-frame) does not obviate this performance advantage.

Deferred Compression Performance. Our final experiment evaluates the performance of deferred compression of uncompressed video writes. To evaluate, we write 3600 frames of the VisualRoad-1K-30% dataset to VFS, leaving storage budget (10× the input, or 2,240MB) and deferred compression configuration (25% threshold) at their defaults. At regular intervals we extract the storage used, Zstandard compression level, and write throughput.

The results for these metrics are listed in Figure 4.14. 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.

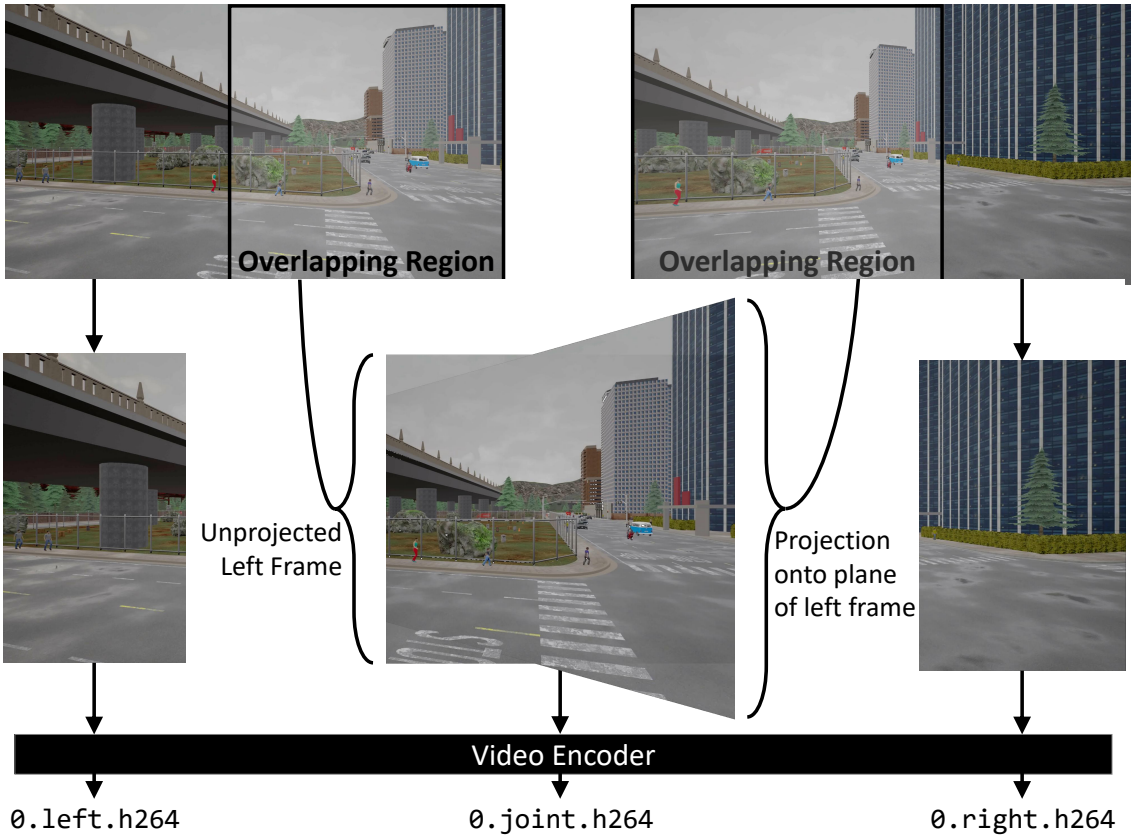
4.6 Summary

In this chapter we presented a video file system (VFS) designed to improve the performance of video-oriented applications and data management systems. VFS decouples high-level operations such as computer vision and machine learning algorithms from the low-level plumbing required to read and write data in a suitable format. Users or higher-level systems leverage VFS by reading and writing video data in whatever format is most useful, and VFS transparently identifies the most efficient method to retrieve that video data. To maximize interoperability, VFS transparently behaves as if it were an ordinary file system.

We compared our VFS prototype against a recent video storage system and a standard local file system. Our experiments showed that, relative to both local storage and other dedicated video storage systems, VFS offers more flexible read and write formats and reduces read time by up to 54%. Our optimizations also decrease the cost of persisting video by up to 45%.

(a) Frame from left video

(b) Frame from right video



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

Figure 4.7: Joint compression applied to two horizontally-overlapping frames. VFS first identifies the overlapping regions in each frame, combines them, and encodes the three resulting pieces (left, overlap, and right) separately. Frames produced using Visual Road [66].

Algorithm 4.1: Joint compression algorithm

let HOMOGRAPHY(f, g) estimate the 3×3 homography matrix of f and g

let PARTITION$(f, [x_0, x_1])$ be the subframe defined by $x_0 \leq x < x_1$

let MERGE(f, g) overlay frame g onto f

function JOINT-COMPRESS(F, G)

Input: Video frames $F = \{f_1, ..., f_n\}$

Input: Video frames $G = \{g_1, ..., g_n\}$

Output: Vector of compressed $(left, merged, right)$ tuples

$C \leftarrow \emptyset$

for $i \in [1..n]$

if $i \mod 30 = 0$ **then**

$H \leftarrow$ HOMOGRAPHY(f, g)

if $H_{1,2} < 0$ **then**

$F, G \leftarrow G, F$

end if

end if

$C \leftarrow C \oplus$ JOINT-COMPRESS-FRAMES(f_i, g_i, H, i)

end for

return C

end function

function JOINT-COMPRESS-FRAMES(f, g, H)

$x_f \leftarrow \left[H^{-1} \cdot \begin{pmatrix} 0 & 0 & \text{WIDTH}(f) \end{pmatrix}\right]_{0,2}$

$x_g \leftarrow \left[H \cdot \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}\right]_{0,2}$

$l \leftarrow$ PARTITION$\big(f, [0, x_f)\big)$ // left part of f

$m_f \leftarrow$ PARTITION$\big(f, [x_f, \text{WIDTH}(f))\big)$ // overlap of f

$m_g \leftarrow$ PARTITION$\big(g, [0, x_g)\big)$ // overlap of g

$r \leftarrow$ PARTITION$\big(g, [x_g, \text{WIDTH}(f_g))\big)$ //right part of g

$m \leftarrow$ MERGE$(m_f, H \cdot m_g)$

return $\big($COMPRESS(l), COMPRESS(m), COMPRESS$(r)\big)$

end function

Algorithm 4.2: Contiguous materialized view compaction

Ensure: All pairs of contiguous physical videos compacted

let HARDLINK(f, d) create hard link for file f in dir d

let INSERT(m, f) insert file f into the temporal index of m

let DIR(m) be the directory associated with m

let VIEWS(v) be the physical videos associated with v

let CONFIG$_{s,p}(m)$ give the spatial/physical config of m

let s_{m_i}, e_{m_i} be the start and end times of m_i

for all $v_1 \in VFS, v_2 \in VFS \setminus v_1$ **do**

  for all $m_1 \in$ VIEWS$(v_1), m_2 \in$ VIEWS(v_2) **do**

    if $e_{m_1}{=}s_{m_2} \wedge$ CONFIG$_{s,p}(m_1) =$ CONFIG$_{s,p}(m_2)$ **then**

      let $F_2 =$ DIR(m_2) sorted by time

      for all $f_2 \in F_2$ **do**

        HARDLINK$(f_2,$ DIR$(m_1))$

        INSERT(m_1, f_2)

      end for

      DELETE(m_2)

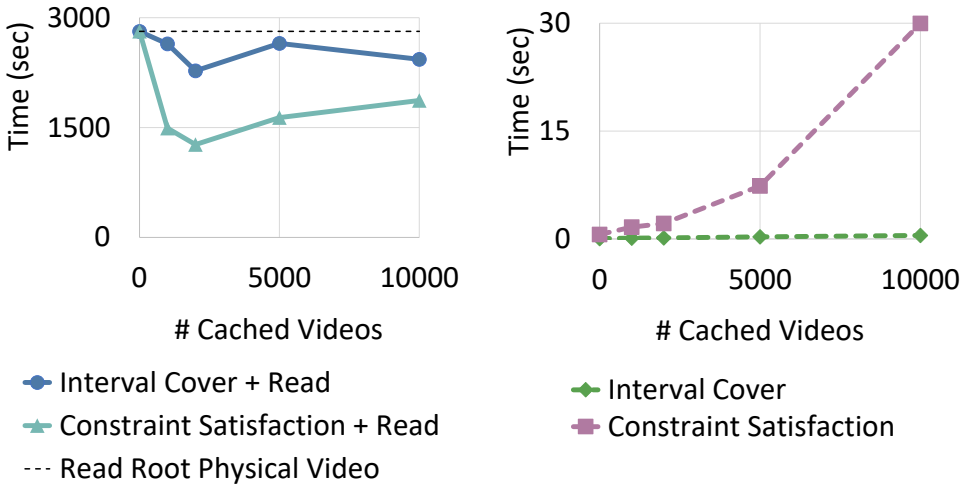
    end

  end for

end for

Table 4.3: Datasets used to evaluate VFS

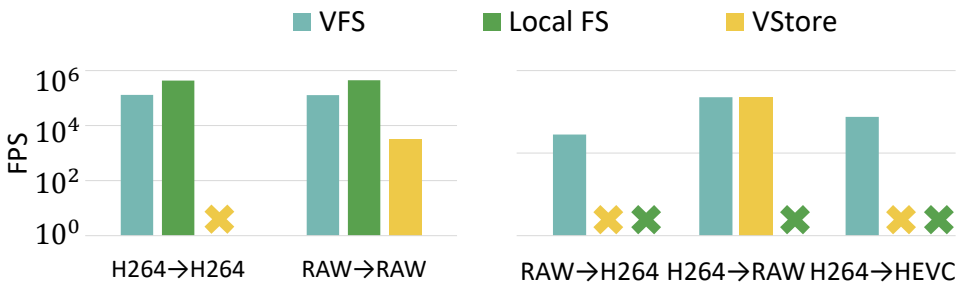
Dataset	Resolution	# Frames	Compressed Size (MB)
Robotcar	1280×960	7,494	120
Waymo	1920×1280	398	7
VisualRoad 1K-30%	960×540	108k	224
VisualRoad 1K-50%	960×540	108k	232
VisualRoad 1K-75%	960×540	108k	226
VisualRoad 2K-30%	1920×1080	108k	818
VisualRoad 4K-30%	3840×2160	108k	5,500



(a) Time to select fragments and read

(b) Time to select video fragments

Figure 4.8: Total time to select fragments and perform a read with varying number of physical videos.



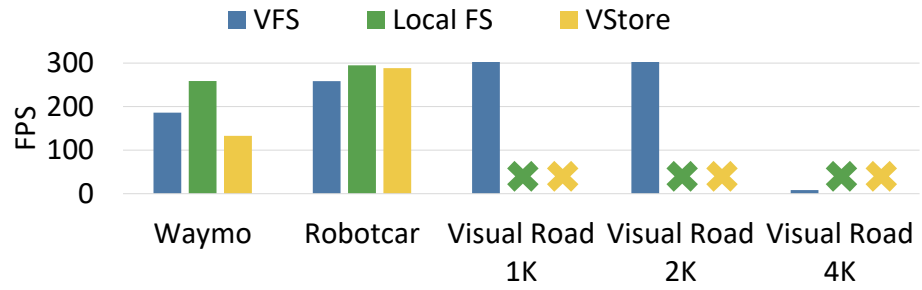
(a) Read in same format (b) Read in different format

Figure 4.9: Read throughput in frames per second for the Visual Road 1K-30% dataset for VFS 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 (e.g., the local file system does not support reading raw video as H264).

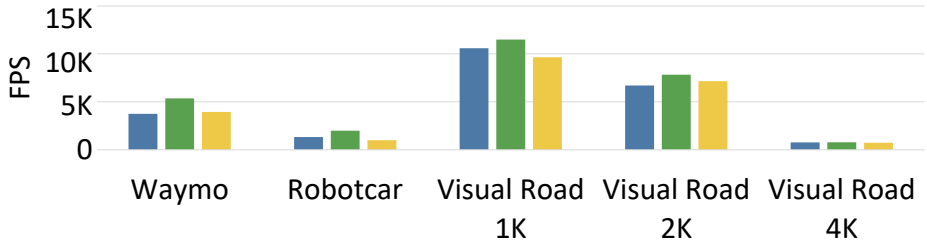
Table 4.4: Joint compression recovered quality

Dataset	Quality (PSNR)	
	Left Frame	Right Frame
Robotcar	62**	17
Waymo	32*	29
VisualRoad-1K-30%	40**	30*
VisualRoad-1K-50%	36*	28
VisualRoad-1K-75%	36*	24
VisualRoad-2K-30%	36*	30*
VisualRoad-4K-30%	36*	30*

** lossless quality * near-lossless quality



(a) Uncompressed writes



(b) Compressed writes

Figure 4.10: Throughput in frames per second to write uncompressed RGB (Figure 4.10(a)) and compressed H264 (Figure 4.10(b)) data to VFS and other baseline systems. An × means that a system does not support the write type (e.g., the local disk lacks capacity to store the >5 terabytes uncompressed VisualRoad-4K-30% dataset).

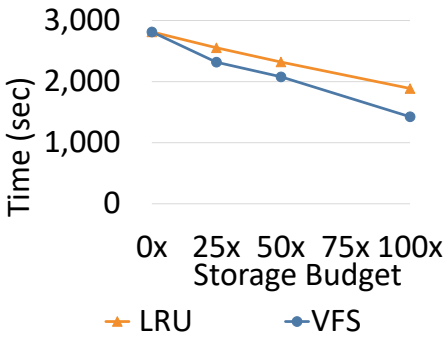


Figure 4.11: Read runtime by storage budget for LRU and VFS cache eviction policies.

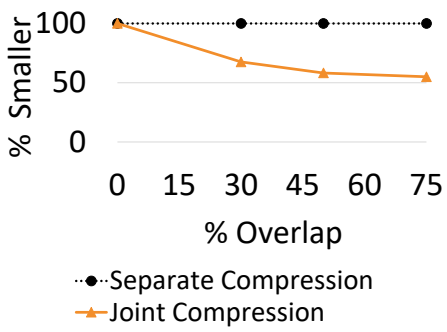
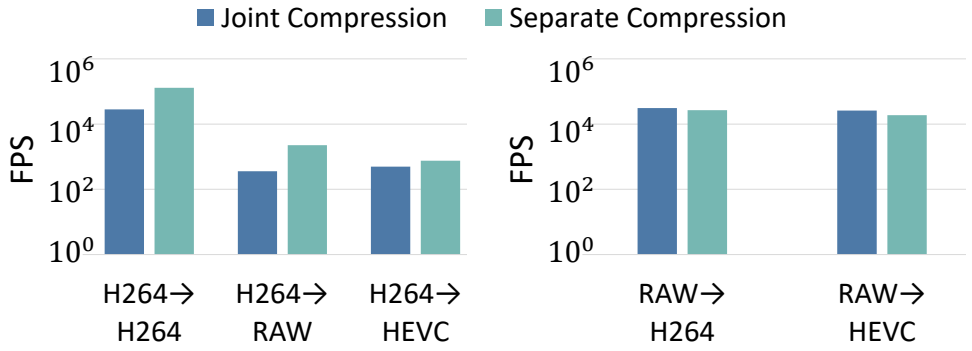


Figure 4.12: Size of joint compression relative to separately-compressed data.



(a) Read throughput (b) Write throughput

Figure 4.13: Throughput for writes with and without the joint compression optimization.

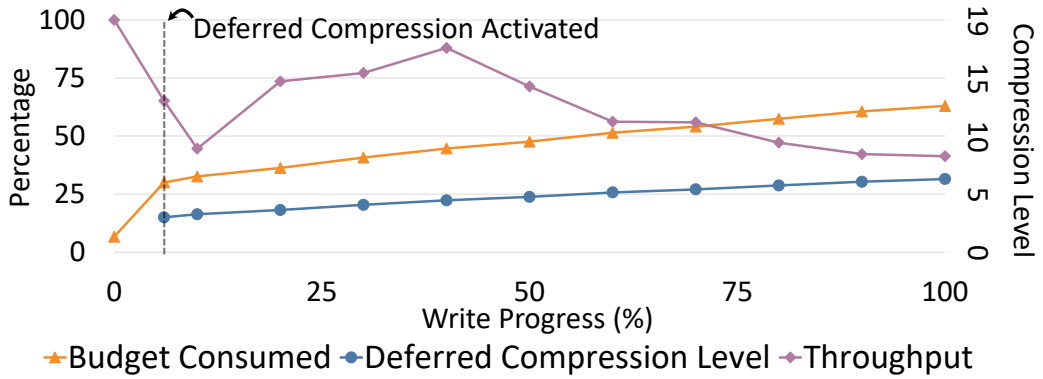


Figure 4.14: Deferred compression performance when writing the VisualRoad-1K-30% dataset to VFS. At 25% the compression optimization activates and changes the slope of the budget consumed.

Chapter 5

VISUAL ROAD: A VIDEO DATA MANAGEMENT BENCHMARK

As detailed in Chapter 1, a number of recent video data management systems (VDBMSs) have been introduced to allow developers to more easily express and efficiently execute video-oriented applications. These VDBMSs quantify their performance by reporting their efficiency when processing various ad hoc workloads both in terms of the input videos selected and the executed queries. However, comprehensive and easily reproducible system comparisons are missing. A key challenge is that there is currently no clear way to reliably and objectively benchmark performance among the various recently proposed VDBMSs. This deficiency is due to a lack of: (i) a robust, sufficiently-complex video dataset (in terms of resolution, quantity, duration, and variety of content); and (ii) an architecture-agnostic specification of a common set of queries that may be executed on current and future VDBMSs. These deficiencies are exacerbated when evaluating visual world applications (VWAs), which require large datasets consisting of many correlated videos capturing the same events from multiple perspectives.

Analogous to standardized benchmarks for other areas of data management research (such as transaction [149] and analytical processing [150]), we develop Visual Road[1], a benchmark designed to evaluate the performance of VDBMSs in the face of a diverse query workload. Visual Road reproducibly and objectively measures how well a VDBMS executes a battery of workloads over a temporal light field (TLF; see Chapter 3) that contain a variety of 2D and 360° videos. Visual Road includes a set of evaluation queries expressed in the TLF algebra

[1]Name inspired by Linear Road [8], a benchmark for streaming database management systems.

(Section 3.2) that a VDBMS executes to evaluate performance. Visual Road is designed to be useful in evaluating VDBMSs that support both VWAs and general video processing applications. The work presented in this chapter appeared in SIGMOD'19 [66].

Visual Road comes with a data generator that populates the TLF used in the benchmark. To allow the creation of a virtually unlimited number of videos embedded in the TLF, Visual Road uses a modern simulation, visualization, and gaming engine [42] to deterministically generate realistic videos within a simulated metropolitan world (see Figure 5.1). Visual Road allows users to vary the city size, number of cameras, and length of video in its simulation to arbitrarily large sizes. Additionally, its simulation allows for the *automatic* calculation of precise ground truth and other metadata about generated videos, without the need for manual annotation. Finally, the cameras used in Visual Road are extremely flexible. In addition to generating ordinary two-dimensional video, they can also produce more complex video types (e.g., panoramic 360° video) that are used with a more complex category of virtual reality (VR)-oriented benchmark queries. Overall, Visual Road's generated video datasets are rich and highly realistic. They can serve to execute various real-world applications such as vehicle tracking and compute meaningful results. The queries provided with the benchmark include a variety of both simple queries and complex applications to exercise benchmarked systems along various dimensions.

Visual Road is designed to be implementable across a wide variety of VDBMS architectures, even those not designed to operate over a TLF. This includes those that perform video querying at scale (e.g., Scanner [124], Optasia [98], Chameleon [83]), operate on emerging forms of video data (e.g., LightDB [65]), and perform deep learning inference (e.g., NoScope [86], BlazeIt [85], Focus [73]). In the same way that relational database systems target subsets of benchmarks (e.g., a specific TPC query), Visual Road is designed to be flexible: a user may either select specific applicable queries or groups of queries appropriate for their systems or execute the entire benchmark to demonstrate broad functionality.

(a) Rain with dense clouds

(b) Overcast skies at sunset

Figure 5.1: Two example frames from Visual Road traffic cameras that illustrate the realism and variety of videos in the Visual Road benchmark. Complete videos available at `visualroad.uwdb.io/datasets`. Some depicted assets copyright [42] and [43].

Visual Road is also extensible, such that future innovations and workload types can be easily incorporated into subsequent versions. This includes both the ability to introduce new and unexpected elements into the TLF, and also to increase the complexity of the benchmark queries (e.g., by increasing the number of cameras, range of benchmark parameters, or available machine learning algorithms).

Each benchmark query is specified in the TLF algebra described in Section 3.2 and is adaptable to a wide variety of VDBMS types and architectures. To illustrate this wide applicability, we have implemented the benchmark on three recent VDBMSs, including the LightDB system described in Chapter 3.

5.1 Synthetic Dataset Generation

The TLF used for the Visual Road benchmark is generated within *Visual City*, a pseudorandomly-generated, simulated metropolitan area. Visual City currently contains road networks, vehicles, pedestrians, landscaping, buildings, bridges, traffic, ground-based cameras, and other features found in real-world cities. Visual City is also affected by a number of conditions such as cloud cover, precipitation, and sun position. Sample photos taken from Visual City are shown in

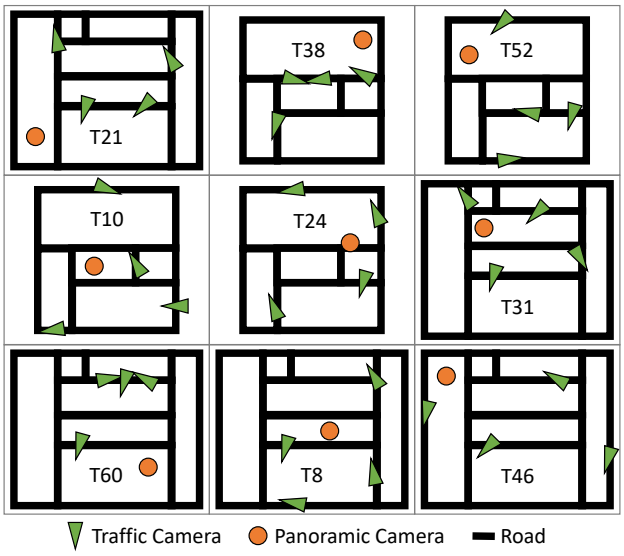


Figure 5.2: Overhead view of a randomized Visual Road TLF with $L = 9$.

Figure 5.1. We posit that this environmental complexity is both important and sufficient to ensure that benchmarked VDBMSs are exercised in interesting ways. The features of the generated city could also be extended in subsequent versions of the benchmark (e.g., by incorporating wildlife, tunnels, or lakes) to increase the complexity, variety, and unexpectedness of the simulation.

5.1.1 Benchmark Data

As shown in Figure 5.2, Visual City is laid out as a disconnected set of tiles. Each tile is drawn uniformly with replacement from a pool of tiles associated with a particular version of Visual Road. The version described in this monograph contains 72 tiles and each tile is several square kilometers in size. Each tile contains different weather conditions, pedestrian and vehicle densities, and geography.

Video data is materialized in the Visual City TLF via a number of cameras. Each tile is associated with a camera configuration C that specifies various types and numbers of cameras. To monitor traffic conditions, each tile contains c_t randomly-oriented *traffic cameras*

positioned 10–20 meters above a roadway, along with c_p randomly-oriented *panoramic cameras* positioned 5–10 meters above sidewalks. Each panoramic camera is composed of four ordinary cameras with overlapping 120° fields of view positioned so that they overlap to capture a 360° field of view. The Visual Road prototype currently sets $C = \{c_t, c_p\} = \{4, 1\}$ for each tile.

When generating the TLF, a user provides values for four hyperparameters. A *scale factor* L determines the number of tiles in the city. A user also selects a *resolution* (e.g., 3840×2160) and a simulation *duration* D that is globally applied to each of the cameras. Finally, a random *seed* s allows other users to deterministically reproduce datasets by reinitializing the pseudorandom number generator with the same seed.

In addition to providing city size, the scale factor also determines the number of queries that a benchmarked VDBMS executes using the Visual City TLF. The Visual Road prototype currently generates $4L$ queries for each type detailed in this section. This batch size allows for a reasonable balance between dataset generation time and benchmark execution time.

The *Visual City Generator (VCG)* is used to generate the TLF containing video captured within Visual City. It accepts the hyperparameters described above and uses these values to construct a Visual City TLF. First, it randomly chooses L tiles from the available set of tiles (with replacement). Each tile is configured and populated using a tile-specific configuration (e.g., pedestrians and vehicles are randomly spawned in number and locations specific to that tile). Cameras are then randomly positioned in each tile subject to the constraints described above. The VCG then executes the simulation and captures videos generated by each camera. These videos are encoded using the H264 codec [144] and stored as flat files. The VCG also generates additional supporting metadata required for verifying the results of specific queries (e.g., Q6 in Section 5.3).

A VDBMS reports performance by executing the benchmark using the Visual City TLF as input. The benchmark comes with a set of pregenerated datasets for immediate use; users may report results using these datasets when comparing to other systems (e.g., "We evaluate using version 1.0 of the 4K-SHORT TLF"). Alternatively, users may deterministically generate

their own datasets (see below) and report the configuration along the results (e.g., "We generated and executed the Visual Road 1.0 benchmark using scale L, resolution R, duration t, and seed s"). By using the same configuration, competing VDBMSs can reproduce the identical dataset and compare results.

5.2 Benchmark Execution

A VDBMS can execute the benchmark either *offline* or *online*. Offline processing simulates batch processing of historical video streams, where the VDBMS has random access to entire video files on persistent storage. Online processing simulates real-time video processing, where data is exposed via a forward-only iterator with unknown total duration.

A separate *Visual City Driver (VCD)* is provided with the benchmark and is responsible for reading the input videos, exposing encoded video data to a VDBMS, submitting queries to the VDBMS being measured, and evaluating the correctness of a VDBMS's query results.

When benchmarking in online mode, a VDBMS may access each video using either a named pipe (on a single local file system) or via the RTP protocol [135]. In this mode, video data is throttled to a simulated real-time throughput (i.e., the VCD exposes video frames at the corresponding camera's capture rate). The VCD blocks on attempts to read video data beyond this rate. For a VDBMS benchmarking in offline mode, the VCD additionally ensures each input video is available on the local file system (on a single node, if the VDBMS supports distributed execution) or a distributed file system.

The VCD uses the scale factor to simulate submission of simultaneous instances (a "query batch") of each benchmark query to the VDBMS. The VCD submits batches in benchmark query order (i.e., Q1 is submitted before Q2). A VDBMS may execute each batch in a manner that is most performant (e.g., serially or in parallel), and may optionally quiesce or restart upon completing a batch.

Each benchmark query is a template with one or more parameters (see Table 5.1). The VCD creates each instance in a query batch by assigning values selected uniformly at random for each parameter from its respective domain. The VDBMS is only responsible for executing the query instance and does not participate in selecting the parameter values.

A VDBMS may do one of two things with the H264- or HEVC-encoded result of a query. First, in *write mode*, as a VDBMS completes each instance in a query batch, it should write the result to a VCD-specified location on the local file system (or on a VCD-specified node for distributed systems) so the VCD can verify the correctness of each output. In this mode the time to persist results is included in the total execution time for the query batch. Alternatively, *streaming mode* allows a VDBMS to discard the results of a query and avoid the write overhead. However, in this mode a user must ensure that the results of the queries are correct, either by executing a second time in write mode or by doing so manually. We show in [66] that the performance differences between these two modes are negligible.

Finally, the VCD also validates the correctness of the results generated by a VDBMS. Depending on the query, it does so either by *frame validation*, which compares VDBMS output videos to reference output videos, or *semantic validation*, which compares a query result with the actual scene geometry used in its input(s). In Visual Road, most microbenchmark queries are verified using frame validation. For these queries, the VCD executes its reference implementation and compares each frame with the VDBMS's output using a *validation metric*. While future versions of Visual Road may allow for different metrics, the one used in the present version is the peak signal-to-noise ratio (PSNR). The PSNR is a frequently-used image comparison metric, and values $\geq$ 40 dB are considered to be near-lossless [78, 71]. Visual Road adopts this threshold as a cutoff for validation.

Query Q2(c) and Q2(d) are verified using semantic validation. In this case, the VCD compares a VDBMS's response to the actual objects that were present in the frames used as input to the query. For example, if a VDBMS indicates a car i is present in frame j, the VCD queries the simulation engine to determine if car i was visible to the camera at the instant the frame was captured.

When reporting results, an evaluator must report validation descriptive statistics for each query. For queries executed in online mode, this should be reported in frames per second. A VDBMS executing offline analytical queries should report total query runtime or frames per second. The evaluator should also report other global elections such as scale factor, resolution, duration, and execution mode.

5.3 Query Suite

The Visual Road benchmark aims to evaluate VDBMS performance by executing a varied workload. It does so by measuring performance using microbenchmark (Table 5.2) and composite macrobenchmark queries (Tables 5.3 and 5.4). Microbenchmark queries target the performance of individual VDBMS operations in isolation. Each microbenchmark involves a single, basic operation exposed by recent VDBMSs that are common in applications that process video—including the visual world applications (VWAs) introduced in Chapter 1. Composite queries, drawn from recent literature (see Chapter 6), are VWAs that utilize two or more microbenchmarks to implement more complex tasks.

A VDBMS individually measures its performance for each query. As detailed previously, for a given query Q_i, the VCD uses the scale factor L to submit a query batch containing $4L$ instances of Q_i to the VDBMS. The free parameters for each instance Q_i^j, summarized in Table 5.1, are uniformly selected (by the VCD) at random from their domain. Below we describe each microbenchmark query. Each query operates on a randomly-selected camera (i.e., a non-null spatial point in the Visual City TLF).

Several queries include ML-based computer vision algorithms, such as object detection. The benchmark requires that all VDBMSs use specified, state-of-the-art algorithms, and focuses on evaluating the execution performance of queries that need to apply those algorithms rather than their quality. For the same reason, the benchmark videos do not purposefully include unusual scenarios designed to challenge computer vision methods. In case query accuracy or algorithm selection becomes a concern, users of the benchmark could be required to publish the F1 scores of their query results.

Table 5.1: Microbenchmark parameters and domains.

Query	Parameter	Domain
Q1	α_1, α_2	$0 \leq \alpha_1 < \alpha_2 \leq 1$
	β_1, β_2	$0 \leq \beta_1 < \beta_2 \leq 1$
	t_1, t_2	$0 \leq t_1 < t_2 \leq D$
Q2(b)	d	$\{10^{-n} \mid n \in [2..4]\}$
Q2(c)	A	YOLO [129]
	$\mathcal{O}$	{Pedestrian, Vehicle}
Q2(d)	m	$\{{}^{n}/_{30} \mid n \in [2..60]\}$
	ϵ	$(0, 1)$
Q3	$\Delta\theta$	$\{{}^{2\pi}/_{2^n} \mid n \in [1..3]\}$
	$\Delta\phi$	$\{{}^{\pi}/_{2^n} \mid n \in [1..3]\}$
	b_i	$\{2^n, n \in [16..22]\}$
Q4, Q5	ρ	$\{2^n \mid n \in [1..5]\}$
	ϕ	$\{2^n \mid n \in [1..5]\}$

5.3.1 Microbenchmarks

The following microbenchmark queries, formally defined using the TLF algebra in Table 5.2, measure a VDBMS's ability to repeatedly perform small operations over input videos. In Section 5.3.2 we compose many of these microbenchmarks to form more substantial, real-world applications drawn from recent literature.

Spatial & Temporal Selection (Q1). A VDBMS must be able to efficiently spatially and temporally select subregions of videos. This ability is exercised, for example, in applications that select highlights containing relevant data, construct cinematographic montages, or apply object detection to a region of interest. Query Q1 measures a VDBMS's ability to perform this type of operation.

Given a cropping region bounded by the horizontal region (α_1, α_2), vertical region (β_1, β_2), and a temporal range (t_1, t_2), query Q1 crops camera data occurring at a random point p_i. The cropping regions and temporal range are chosen uniformly at random.

Transformation (Q2) & Subquery (Q3). A VDBMS must be able to efficiently perform transformations at various granularities (e.g., per-pixel, using a stencil, over regions, and for entire frames). Queries Q2 and Q3 test a VDBMS's ability to transform input videos at these scales.

The first transformation, Q2(a), requires that a VDBMS convert video data captured at location p_i to grayscale. The VCD reference implementation does this by dropping chroma information (i.e., the U and V channels in YUV color space) and leaves luminescence (Y) unchanged.

Query Q2(b) performs a Gaussian blur convolution [133] over an input video by applying a $d \times d$ kernel over a video at spatial position p_i. It does so by invoking a user-defined function *blur* that is parameterized by the kernel size.

Next, query Q2(c) generates rectangular bounding boxes for objects in a video captured at location p_i. It does so by applying an object-detection algorithm A to each input video frame (in the present version A is a singleton consisting of the YOLO [129] algorithm). This algorithm associates each pixel x_j with zero or more object classes $\mathcal{O} = \{o_1, ..., o_n\}$. The VDBMS associates a constant color c_k with each class o_k and a "null" black color ω for regions not associated with any class. It finally produces an output video with frames containing pixels given by:

$$x'_j = \begin{cases} c_{\min \mathcal{O}} & \text{when } \mathcal{O} \neq \emptyset \\ \omega & \text{otherwise} \end{cases}$$

Q2(c) is verified using semantic validation, where each detected object is mapped back to an actual object in the scene geometry that produced the input video, and a reference bounding box is generated for the object. The maximum Jaccard distance between the VDBMS-generated and reference boxes must not exceed ϵ. In the prototype version of Visual Road we have adopted the PASCAL VOC [45] threshold of $\epsilon = 0.5$.

Query Q2(d) performs background masking on each input video by applying a mean filter [9] to each video frame. Background masking is useful for removing static, unchanging regions of a frame (e.g., sidewalks and buildings) and leaving the dynamic "foreground" areas untouched. For each window of m video samples $f_j, ..., f_{j+m}$ in a video, a VDBMS should compute a background reference frame $b_j = \frac{1}{m} \sum_{k \in [i..i+m]} f_k$. Next, for each pixel p_v in frame f_j and p_b in background reference frame b_j, the VDBMS should output a black pixel ω when their difference is below the threshold $|\frac{p_v - p_b}{p_v}| < \epsilon$ and p_v otherwise.

Finally, query Q3 performs an operation on individual regions of the Visual City TLF. For example, an application might deliver less-important regions at lower bitrates (see Q10) or blur regions of a video frame that contain faces or other sensitive information. Q3 performs the former operation by segmenting a video at point p_i into regions of size $(\Delta\theta, \Delta\phi)$. Each resulting region u_i is re-encoded at a bitrate given by b_i. The resulting regions should then be recombined.

Interpolation & Resampling (Q4, Q5). Computer vision algorithms and machine learning models frequently require an input image sampled at a particular resolution. These queries test a VDBMS's ability to perform this sampling by asking it to perform interpolation and resampling operations on input videos. First, query Q4 increases a video's resolution to $(\rho R_x, \phi R_y)$ using bilinear interpolation. The resolution $R = (R_x, R_y)$ is drawn from the hyperparameters used to generate the dataset. Query Q5 performs the inverse operation: given a video at point p_i, the VDBMS downsamples each frame to a lower resolution $(\frac{R_x}{\rho}, \frac{R_y}{\phi})$.

Union (Q6). Modern video applications frequently require combining two or more data streams. For example, an augmented reality application might overlay advertising or informational text on a user's display. Queries Q6(a) and Q6(b) test a VDBMS's ability to perform these operations by merging and combining data stored in various formats.

In particular, query Q6(a) merges the video at point p_i with a *bounding box video* $b = Q_{2c}(p_i)$ by performing an outer join on the corresponding pixels within each video. The bounding box video is generated offline by the VCD by applying the reference implementation of query Q2(c) to the associated input video. The VCD exposes b in two formats: as an

encoded video and as a serialized sequence of bounding box class identifiers and coordinates. VDBMSs may consume either format when executing the query. As in Q2(c), the VCD uses the black sentinel color ω to represent null pixels in the encoded variant of b.

Query Q6(b) overlays a set of text annotations stored in the TLF C onto an input video. Here the data at point p_i in C is a WebVTT [123] file embedded as a metadata track within the input video's container. The VCD randomly generates the WebVTT file and randomly varies position and non-overlapping duration of each annotation. Benchmarking VDBMSs may render the annotations using any font, and need only support the `line` and `position` cue settings.

5.3.2 Composite Benchmarks

This section describes more complex, real-world workloads that we call *composite benchmarks*. Each composite benchmark leverages one or more of the microbenchmarks introduced in the previous section. Composite benchmarks are drawn from recent examples and applications in the computer vision and machine learning literature (see Chapter 6).

Object Detection (Q7). This query leverages Visual City cameras to identify instances of a given object class $o \in O$ (e.g., pedestrians or vehicles). To draw attention to identified objects, it also removes extraneous "background" portions of each video frame that do not contain visual information about the class and persists or streams the results.

At a high level, a VDBMS implementing this query first applies the classification query Q2(c) to every non-null point in the Visual City TLF V. Next, for each object type, it overlays the resulting bounding boxes onto the video data using query Q6(a). Finally, it refines the results by performing background removal as defined in Q2(d). Figure 5.3 illustrates this process applied to a single video frame.

As formalized in Table 5.3, the inputs to the object detection query is the TLF V, and object detection is applied to traffic cameras located as points $t_1, ..., t_{c_t \cdot L}$. A VDBMS may report results using additional object classes or detection algorithms, so long as it also includes results for those defined in Table 5.1.

(a) Input video frame

(b) Output video frame

Figure 5.3: Sample input and output for one frame of the object detection query (Q7). Some depicted assets copyright [42] and [43].

Vehicle Tracking (Q8). Query 8 simulates the tracking of vehicle sightings throughout Visual City. Each automobile in Visual City has a unique front-facing license plate containing six random alphanumeric digits.

A vehicle sighting instance is defined by the period in which it is identifiable by one or more traffic cameras. Initially, a vehicle *enters* a traffic camera's field of view when its license plate is unobscured relative to that camera. It *exits* the traffic camera's field of view when one or both of these conditions is no longer met. The video frames occurring between a vehicle entering and exiting a camera is a *vehicle tracking segment* (VTS).

The VCD simulates searching for vehicles by issuing *vehicle tracking queries* to the VDBMS. The input to this query is the license plate of a random vehicle. As illustrated in Figure 5.4, the output is a *tracking video* of temporally-ordered (by entry time), concatenated VTSs for the vehicle associated with that license plate.

This query is formalized in Table 5.4 as a recurrence. Its output is defined by repeated application of Q2(c). Each application uses a license plate recognition function $\mathcal{L}$ to identify the next VTS_i in the input video. Query Q1 is used to select the temporal range $[t_i, t_{i+1}]$ and the output is appended to the previous iteration until a fixpoint is reached.

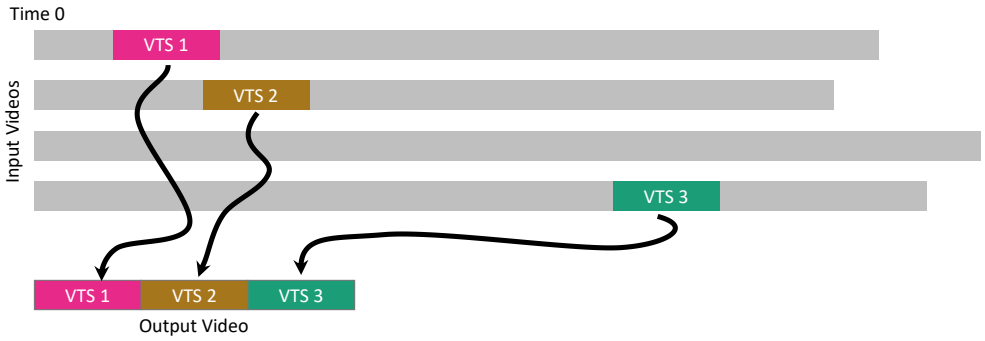


Figure 5.4: Illustration of a vehicle tracking query on a Visual Road dataset.

Illustration of the vehicle tracking query (Q8) on a Visual Road dataset (scale = 1) that contains three vehicle tracking segments (VTSs). Each VTS is temporally ordered, concatenated, and output.

5.3.3 Virtual Reality

Virtual reality (VR) video is an important, emerging subclass of video data. Panoramic VR videos (a.k.a. 360° videos) are one popular member of this subclass. Visual Road includes two benchmark queries that target the VR 360° data format. We include these queries because VR video operations exercise sophisticated features possible only in the most recent VDBMSs, and evaluating their performance at scale is an important differentiating factor between such systems. Operations on VR videos are also useful to test a VDBMS's ability to use higher resolutions than typically seen in ordinary 2D video.

The following queries are more open-ended than the previous benchmark categories, allowing an implementing VDBMS additional freedom to optimize their execution.

Panoramic Stitching (Q9). Modern panoramic cameras produce video panoramas by "stitching" together two or more ordinary 2D videos into a 360° video. To take advantage of modern video compression, the spherical video is reprojected onto a plane and encoded as if it were an ordinary 2D video. Query Q9 requires that a VDBMS perform this process by stitching video data from the panoramic cameras scattered throughout Visual City. Recall

from Section 5.1 that each panoramic camera is composed of four ordinary 2D cameras with a 120° field of view. A VDBMS implementing Q9 should accept the video streams from the constituent 2D cameras, execute a function to convert the four images into a single 360° video, and output it. This process should be repeated for every panoramic camera in Visual City.

A VDBMS is free to implement the conversion in any manner that is most efficient, with the constraints that (i) the resulting 360° videos should be equirectangularly projected [133] and (ii) the result should be moderately similar (i.e., within 30 dB PSNR) to the reference implementation.

Tile-Based Encoding (Q10). Recent research has suggested that streaming "unimportant" areas of a 360° video in lower resolution may yield substantial bandwidth savings and reduced storage sizes [72, 168, 67, 108]. Additional savings may be achieved by reducing the resolution of a VR video to match the resolution of the VR headset or viewing device. This query, formalized in Table 5.5, measures a VDBMS's ability to use both techniques to reduce bandwidth costs. To execute this query, a VDBMS should use Q3 to decompose each video frame into nine equal-sized "tiles" and encode high-importance tiles at a high-quality bitrate b_h and the remaining tiles at a low-quality bitrate b_l. The VDBMS should also use Q5 to downsample the video to a lower resolution that matches the viewing device. For simplicity, we treat these parameters as global values that are applied over the entire duration of the input 360° video.

5.4 Implementation

We implement the video generators for Visual Road 1.0 by adapting CARLA 0.84 [42], an open-source simulator designed for autonomous driving research. CARLA itself is designed as a "plugin" for the Unreal Engine 4.18, a commercial gaming engine that provides physics, simulation, and other graphics-oriented features. CARLA includes resources, textures, and geometry, which form the basis of the tiles used in Visual Road. It also exposes a configuration-driven API that facilitates camera placement, rendering, and other convenience

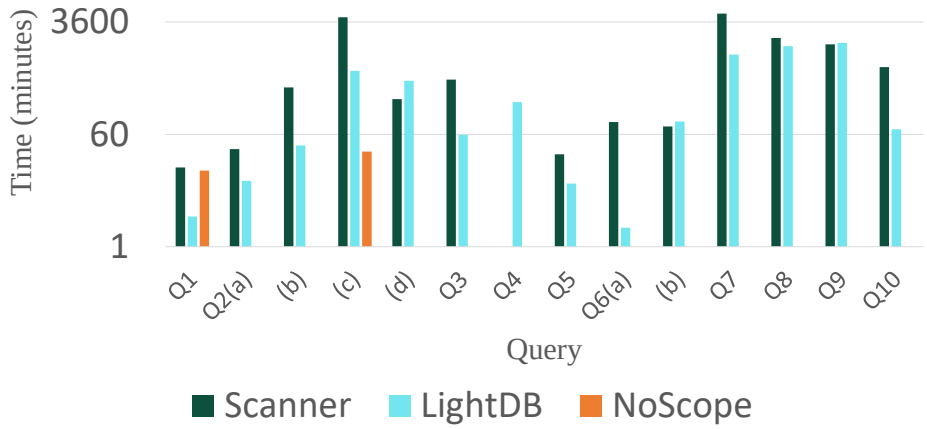
functionality. We modify CARLA to support efficient video encoding, camera rendering at varying resolutions and frame rates, and multiple tiles and configurations. All artifacts are developed using C++.

Version 1.0 of Visual Road contains a tile pool consisting of 72 tiles. Each tile is constructed from one of two maps (TOWN01 and TOWN02), both drawn from the set of CARLA resources. Each is also associated with one of twelve different weather configurations and one of three different vehicle and pedestrian densities (e.g., a "rush hour" tile contains 120 vehicles and 512 pedestrians). Each tile is configured with 4 traffic cameras and 1 panoramic camera, both capturing at 30 frames per second.

We also develop a Visual Road reference implementation for use in verifying benchmark results using PSNR comparisons. The reference implementation was written in Python and depends on OpenCV [118] for video-related operations. For semantic verification, the reference implementation interacts with the Unreal Engine to generate metadata relating to objects in a camera's frame of view.

The current version of Visual Road includes support for H264 [160] and HEVC [144], and each query result must be encoded using either of these codecs. Visual Road also currently supports frame rates in the range of 15-90 frames per second (FPS) and resolutions at 1K (960×540), 2K (1920×1080), and 4K (3840×2160). However, we anticipate that future versions will extend support to additional codecs, containers, frame rates, and resolutions.

The VCG supports single-node and distributed modes of input video generation. In distributed mode, the VCG uses the Amazon Elastic Compute Cloud (EC2) to launch multiple instances of the Unreal Engine in parallel. Each node independently configures the Visual Road environment, launches an Unreal Engine instance, simulates the tile for which it is responsible, and collects video output.



(a) Runtime

Figure 5.5: Visual Road log-scale performance by query with scale factor $L = 4$ at 1K and 60 minutes.

5.5 Evaluation

We experimentally evaluate Visual Road in three ways. First, in Section 5.5.1, we demonstrate that Visual Road produces performance results for VDBMSs similar to real-world datasets and better than alternative synthetic approaches. Next, in Section 5.5.2, we apply the benchmark to three recent open-source VDBMSs and contrast the results. For these experiments we show out-of-the-box performance numbers for all of the experiments. Better results could certainly be achieved for each system with appropriate tuning. Our goal is to evaluate the benchmark and not the systems. Next, in Section 5.5.3 we evaluate the quality of the video generated by Visual Road, and in Section 5.5.4 we evaluate the performance differences between write and streaming execution modes. Finally, in Section 5.5.3 we evaluate the scalability of Visual Road when generating large corpora.

Experimental configuration. Except where stated otherwise, we perform experiments using a hardware configuration consisting of a single physical machine running Ubuntu 16.04 and containing an Intel 3.4 Ghz i7-6800K processor with 6 cores and 32 GB RAM. It is equipped with a 256 GB SSD drive and a Nvidia Quadro P5000 GPU.

Benchmarked VDBMSs. To show wide applicability, we execute Visual Road on three recent, open-source VDBMSs: LightDB [65] (see Chapter 3), Scanner [124], and NoScope [86]. These VDBMSs cover a variety of target use-cases, respectively including high-performance virtual and augmented reality video, analytics at scale, and specialized application of deep learning models.

Scanner is an open-source VDBMS that offers efficient distributed video processing at scale. We deploy Scanner using its most recently-published Docker container, which was built using CUDA 8.0 [114], OpenCV 3.2 [118], and FFmpeg 3.3.1 [17]. Scanner lacks support for video cropping (Q1), captioning (Q6(b)), and license plate recognition (Q8), so we add these features as custom C++ operators (respectively) using a modified resize operator, the `libwebvtt` [111], and `libopenalpr` [117]. We also make minor modifications to Scanner's grayscale and resizing kernels so that queries Q2(a) and Q4–5 can be expressed.

NoScope is a specialized VDBMS that improves the performance of applying deep learning models to video at scale. We deploy the most recent prototype of NoScope, which relies on TensorFlow 0.12, CUDA 8.0, and cuDNN 5.1. Because NoScope is specialized for deep learning and does not expose support for arbitrary queries or a mechanism for extensibility, we are only able to express queries Q1 and Q2(c) using this system.

We execute all queries in this section using VCD's offline mode, since neither Scanner nor NoScope support operating on live-streaming video data. Except where stated otherwise, for all systems we use default settings and did not attempt to optimize batch size or leverage other optimization strategies.

5.5.1 Dataset Validation

In this section, we evaluate whether Visual Road's synthetically generated data yields performance numbers similar to using real videos. We also evaluate whether other types of synthetic inputs could work as well as Visual Road to test a VDBMS. Overall, we find that Visual Road-generated input videos produce runtimes similar to using real-world, manually-annotated data, whereas other synthetic datasets may yield misleading or incorrect results.

As a real-world baseline, we use the UA-DETRAC [159] video dataset. UA-DETRAC is a manually-annotated corpus composed of recorded traffic camera videos of various durations. Our experiments in this section utilize the training subset, which consists of 60 sequences of 1K video recorded at 25 FPS. The data are provided as 83,791 images, which we H264-encode to produce approximately two hours of video.

We next use Visual Road 1.0 to create input videos that match the UA-DETRAC configuration. We execute the VCG with scale factor $L = 16$ at 1K resolution to produce 64 traffic camera videos. From these, we randomly select 60 videos and re-encode each to 25 FPS. We finally truncate each video so its duration matches a corresponding video in the UA-DETRAC dataset.

In addition to comparing with the Visual Road-generated input videos, we also construct two alternative synthetic datasets as follows:

Duplicate videos. A user might test a VDBMS by reproducing one or more manually-annotated videos to create a larger synthetic corpus. To simulate this process, we select the longest UA-DETRAC video ("MVI_40172") and replicate it 60 times. We then truncate each replicated video to match the duration of a corresponding video in the UA-DETRAC dataset.

Random videos. Alternatively, a user might use randomly-generated video to evaluate VDBMS performance. To simulate this approach, we generate a fully-synthetic video corpus consisting of random noise. As in the previous dataset, we generate 60 videos matched in duration to UA-DETRAC.

We execute the microbenchmark queries on the Scanner and LightDB VDBMSs using each of the datasets described above. Because NoScope is only able to execute two of the queries, we omit it from this experiment. We were not able to execute Q4 on Scanner for reasons we describe in Section 5.5.2.

Table 5.6 shows the performance results for each VDBMS, query, and dataset. For each of the queries, VDBMS performance for the Visual Road input is similar to the UA-DETRAC input. In no cases does the Visual Road dataset lead to a result that disagrees with the UA-DETRAC counterpart, meaning that the benchmark correctly identifies the faster system for each query. In general, performance for each query closely tracks the baseline.

The duplicate and random datasets do not consistently agree with the UA-DETRAC performance results. For these datasets, at least one query produces a result that disagrees with the baseline (i.e., where system X performs faster than system Y on UA-DETRAC but worse on the synthetic input) and could lead a user to draw an incorrect inference about the performance of a system when using real-world video.

Equally problematic are the cases where the performance differences between systems differ by more than an order of magnitude compared to the baseline dataset. We have highlighted discrepancies of this magnitude on Table 5.6. Such a difference occurs for more than one query in both the duplicate and random datasets and could lead a user to draw an incorrect conclusion about the relative performance differences between VDBMSs when using one of these synthetic datasets.

Overall, system performance on Visual Road data is similar to the real videos with the important advantage that Visual Road data is synthetically generated and videos can thus be scaled and parameterized as needed.

5.5.2 System Comparison

In this section, we apply the benchmark to the comparison VDBMSs at various scale factors and show that Visual Road is a useful benchmark for comparing performance between systems. The time to generate a Visual Road dataset need only be incurred once since a given configuration determines the resulting videos.

Our first experiment gives a high-level overview of VDBMS performance. In this experiment, we hold constant the scale factor ($L = 4$), resolution (1K), and duration (1 hour). We use this configuration and execute applicable benchmark queries on each VDBMS.

Figure 5.5 shows the log-scale total runtime for each system and query combination. NoScope shows excellent performance on Q2(c)—which closely matches the workloads it was designed to execute—but its highly specialized implementation doesn't support most of the other benchmark queries. Scanner and LightDB show similar performance on Q1, Q6(b), and Q7–Q10.

We next vary the scale factor L while holding other parameters constant at their previous values. To accomplish this, we used the VCG to generate a series of one-hour datasets at 1K resolution with increasing the size of the simulated city. We then execute each query on a VDBMS and measure the total runtime until completion. As we discussed previously, the NoScope system only supports Q1 and Q2(c) and so we show results only for these queries.

Figure 5.6 shows detailed VDBMS performance for each benchmark query. At small scale factors, no single system dominates across all queries. As the scale factor increases, however, Scanner often falls behind the other comparison systems. This drop-off appears to be due to memory thrashing as more video data are introduced. Scanner also suffers from a poorly-performing resize kernel (Q1) and its use of the Caffe [82] deep learning framework to execute the Q2(c) YOLO neutral network.

LightDB performs well across many queries but suffers from a CPU-only implementation of the captioning query. As expected, NoScope excels at efficiently applying the YOLO neural network in query Q2(c).

Both Scanner and LightDB have memory-related issues when executing Q4. When we execute this query on Scanner, it quickly allocates all available memory and thereafter fails to make progress. This occurs even when we attempt to execute Q4 on Scanner with one input video or with a custom, Python-based implementation of the resize operator.

LightDB exhibits similar issues when attempting to subquery (Q3) or resize (Q4) more than 40 videos, after which it fails due to lack of GPU memory. We work around this by issuing these queries in two batches — one with the first 40 videos, and a second with the rest.

We also observe that the composite and VR benchmark queries took far longer for both systems than did the microbenchmark queries (with the exception of Q2(c), which requires executing an expensive convolutional neural network). This supports that Visual Road is effectively targeting a wide range of workload complexities.

Our final comparison shows the lines of code (LOC) required to execute each query on a VDBMS. To calculate LOC, we construct a file containing the minimal code required to execute each query, auto-format it, and count the number of non-empty lines. Scanner and NoScope expose Python bindings and we use AUTOPEP for formatting; we similarly use CLANG-FORMAT for LightDB's C++ API. We separately count implementation for queries that required additional logic (e.g., LightDB's text caption plugin for Q6(b)) using the same approach.

Figure 5.7 shows the resulting counts. Here, Scanner and LightDB have similar LOC counts for many queries. The same is true for supporting extension implementation, primarily because both are written in C++. Because NoScope narrowly targets only a single query, invoking it requires only a few lines of Python code.

Overall, Visual Road effectively shows that NoScope is an excellent, highly specialized engine while Scanner and LightDB are more general purpose. It also exposes the performance advantages and limitations of each system on the different query types.

5.5.3 Video Quality & Generation Time

Quality of Video

In this section we examine the quality of video produced by Visual Road and how similar it is to real video. Again, our goal is to provide evidence that videos are of sufficiently good quality to be used to evaluate query execution time.

To evaluate this aspect, we use the YOLOv2 [129] model to identify automobiles (i.e., cars and vans) in both synthetic Visual Road and real UA-DETRAC video. This model comes pretrained on the COCO training/validation dataset [93]. Each test set contains 1920 randomly-selected frames.

The average precisions (APs) at 50% IoU for the Visual Road and UA-DETRAC datasets were respectively 72 and 75%. This is similar to results reported by Redmon & Farhadi (AP = 77% [129]) for this model on the "car" category of another benchmark dataset [45]. This suggests that the semantic structure of the synthetic Visual Road video is similar to that of real video and supports its use for evaluating the query execution time of a VDBMS at scale. However, these results notwithstanding, we again emphasize that Visual Road is not intended to train machine learning models or evaluate a VDBMS in terms of prediction accuracy.

Generator Performance

We next explore the performance of the Visual Road Generator (VCG) when creating large video datasets. Figure 5.8 shows the total time to generate a one-hour dataset with increasing scale factor and at three resolutions: 1K, 2K, and 4K. For this experiment, we executed the VCG on a single node using the hardware configuration described previously.

These results show an approximately linear increase in single-node generation time as the scale factor increases. This result is intuitive, since (i) the number of cameras is a linear function of scale factor, (ii) at a constant resolution the total number of generated pixels increases linearly with number of cameras, and (ii) the underlying scene geometry must be

recalculated on a per-camera basis, precluding opportunities to render in parallel. The 4K generation time increases more rapidly due to a software limitation related to the number of cameras that can be simultaneously instantiated; we plan to further optimize this in the future.

We next evaluate the performance of the VCG in distributed mode when generating video in parallel using multiple nodes. We hold constant scale factor ($L = 2$), resolution (1K), and duration (1 hour), and vary the number of nodes used to execute the VCG. For this experiment, we use `p3.2xlarge` nodes on the Amazon Elastic Compute Cloud (EC2) to generate video in parallel. Each instance has one Nvidia V100 GPU, 8 logical cores, and 61GiB of RAM.

Figure 5.9 shows the time required by the VCG to generate a dataset with the above configuration and given number of nodes. Because dataset generation does not require coordination between cameras, we see an expected linear decrease in generation time as we increase the number of nodes available for processing.

5.5.4 Write & Discard Modes

Our final set of experiments evaluate the performance differences between benchmark execution in write and streaming modes (see Section 5.2). To do so, we executed the benchmark on the Scanner and LightDB systems in each execution mode. To support streaming mode on Scanner, we modified each query to send results to the null device. We used LightDB's sink operator for this operation.

For each query, we found that the performance difference between the two modes was less than 2.5%. This difference is in part due to pipelineing and also because disk IO is inexpensive relative to video compression.

5.6 Summary

In this chapter we presented Visual Road, a benchmark for video data management systems (VDBMSs). Visual Road comes with a data generator that produces an unlimited amount of synthetic video generated by simulating an active metropolitan area, along with a suite of queries that evaluate VDBMS performance.

Our results show that video generated using Visual Road closely matches real-world, manually-annotated video corpora and allows VDBMSs to be evaluated at any scale. We used an implementation of the Visual Road benchmark to evaluate the performance of several modern VDBMSs and show that it is a useful tool for capturing meaningful performance comparisons between systems.

Table 5.2: Microbenchmark queries over the TLF V expressed using the algebra described in Section 3.2. Each query i targets a camera at location $p_i \in volume(V)$ in Visual City.

#	Name	Pseudocode
Q1	**Select**	$\textsc{select}\Big(\textsc{select}\big(\textsc{scan}(V), p_i\big), \alpha \in (\alpha_1, \alpha_2), \beta \in (\beta_1, \beta_2), t \in (t_1, t_2)\Big)$ Select video data at point p_i and crop it between (α_1, α_2), (β_1, β_2), and time (t_1, t_2).
Q2	**Transform**	
(a)	Grayscale	$\textsc{map}\Big(\textsc{select}\big(\textsc{scan}(V), p_i\big), f\Big)$ Convert a video to grayscale using f that takes in a YUV pixel (y, u, v) and returns $(y, 0, 0)$.
(b)	Blur	$\textsc{map}\Big(\textsc{select}\big(\textsc{scan}(V), p_i\big)$,blur(d), $R_d\Big)$ *blur* generates a $d \times d$ Gaussian convolution function which is applied to the video data at p_i. The hyperrectangle R_d is defined by $(x, y, z, t) = \mathbf{0}, \theta \in (-d, d), \phi \in (-d, d)$.
(c)	Boxes	$\textsc{map}\Big(\textsc{select}\big(\textsc{scan}(V), p_i\big), boxes(A, O), R_{\alpha\beta}\Big)$ *boxes* returns a function that identifies object classes O using algorithm A. It is applied to the video at p_i to produce boxes for detected instances. The hyperrectangle $R_{\alpha\beta}$ is defined by $(x, y, z, t) = \mathbf{0}, \theta \in (0, 2\pi), \phi \in (0, \pi)$.
(d)	Masking	$v = \textsc{select}\big(\textsc{scan}(V), p_i\big)$ $b = \textsc{map}(v, mask, R_m)$ $\textsc{union}\left(v, m, p \mapsto \begin{cases} \omega & \text{when } \lvert v(p) - b(p)\rvert < \epsilon \\ v(p) & \text{otherwise} \end{cases}\right)$ Apply an m-frame mean-filter at p_i, and set non-null values below threshold ϵ to the mean. *mask* is a function that returns the mean color in the range R_m. The hyperrectangle R_m is defined by $(x, y, z, \theta, \phi) = \mathbf{0}, t \in (0, m)$.
Q3	**Subquery**	$\textsc{subquery}\Big(\textsc{partition}\Big(\textsc{select}\big(\textsc{scan}(V), p_i\big), \Delta\theta, \Delta\phi\Big), reencode(b_1, ..., b_n)\Big)$ Cut each video frame into n components of size $(\Delta\theta, \Delta\phi)$ and use the function *reencode* to encode tile i at bitrate b_i.
Q4	**Upsample**	$\textsc{discretize}\Big(\textsc{interpolate}\Big(\textsc{select}\big(\textsc{scan}(V), p_i\big), bilinear\Big), \rho R_x, \phi R_y\Big)$ Upsample video data to $(\rho R_x, \phi R_y)$ samples using bilinear interpolation. $R = (R_x, R_y)$ is the Visual Road resolution hyperparameter described in Section 5.1.1.
Q5	**Downsample**	$\textsc{discretize}\Big(\textsc{select}\big(\textsc{scan}(V), p_i\big), {}^{R_x}/_{\rho}, {}^{R_y}/_{\phi}\Big)$ Reduce each video to $(\frac{R_x}{\rho} \in \mathbb{N}, \frac{R_y}{\phi} \in \mathbb{N})$ samples. $R = (R_x, R_y)$ is the Visual Road resolution hyperparameter (see Section 5.1.1).
Q6	**Union**	
(a)	Boxes	$b = Q_{2c}(p_i)$ $\textsc{union}\Big(b, \textsc{select}\big(\textsc{scan}(V), p_i\big), left\Big)$ Overlay bounding rectangles b produced by query Q2c on top of video data at p_i.
(b)	Captions	$\textsc{select}\Big(\textsc{union}\big(\textsc{scan}(C), \textsc{scan}(V), left\big), p_i\Big)$ Overlay captions defined in C_i on top of an input video V_i.

Table 5.3: Object detection query (Q7).

Input	TLF V
	Traffic camera positions $p_1, ..., p_{c_t \cdot L}$
	Object detection function $A(V, O)$
	Object classes $O = \{o_1, ..., o_m\}$
Output	Videos $\{v_1^{o_1}, ..., v_n^{o_m}\}$ where
	$v_i = \text{SELECT}(V, p_i)$
	$v_j^{o_i} = Q_{2d}\Big(Q_{6a}\big(v_i, Q_{2c}(v_j, A, \{o_i\})\big)\Big)$

Table 5.4: Vehicle tracking query (Q8)

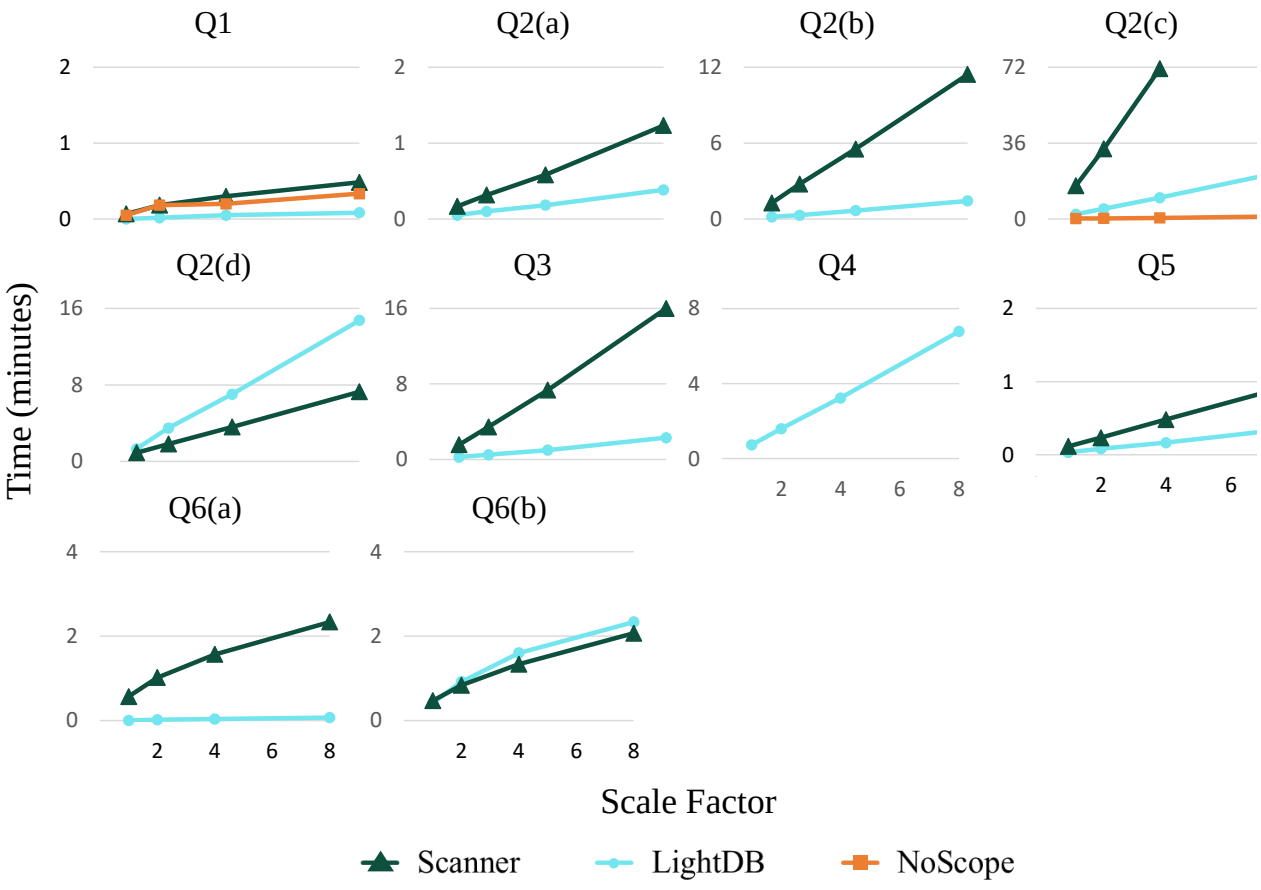
Input	TLF V
	Traffic camera positions $p_1, ..., p_{c_t \cdot L}$
	License plate $l = (l_1, ..., l_6)$
	License plate recognition function $\mathcal{L}$ (OpenALPR)
Output	Video $v_{\text{out}} = VTS_1 \oplus \cdots \oplus VTS_n$ where
	$v_i = \text{SELECT}(V, p_i)$
	$t_j = \sum_{k \in [1..j-1]} \lvert volume(VTS_k).time \rvert$
	$VTS_i = Q_1\Big(Q_{6a}\big(v_i, Q_{2c}(v_i, \mathcal{L}, \{l\})\big), (t_i, t_{i+1})\Big)$

Table 5.5: Tile-Based streaming query (Q10).

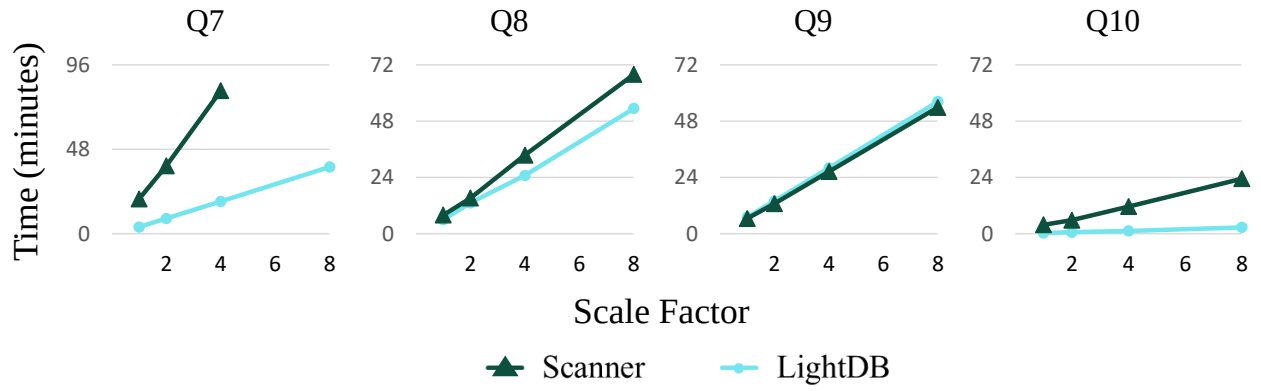
Input	TLF V
	360° positions $p_1, ..., p_{c_t \cdot L}$
	Bitrates $B = (b_1, ..., b_9), b_i \in \{b_h, b_l\}$
	Client resolution $R_c = \{r_1, ..., r_n\}$
Output	Videos $\{v'_1, ..., v'_n\}$ where
	$v_i = Q_9\big(\text{SELECT}(V, p_i)\big)$
	$V'_i = Q_5\big(Q_3(v_i, j \rightarrow b_j), r_i\big)$

Table 5.6: Visual Road ability to accurately measure VDBMS performance compared with real videos. Values show total runtime in minutes and speedup relative to the UA-DETRAC baseline for the LightDB system described in Chapter 3 and Scanner. Red cells indicate a result where the relative performance between systems differs from the baseline, while Yellow cells show performance discrepancies of an order of magnitude or more relative to the baseline.

Query	UA-DETRAC		Visual Road		Duplicates		Random	
	LightDB	Scanner	LightDB	Scanner	LightDB	Scanner	LightDB	Scanner
Q1	1	2	1 (0.9×)	2 (0.8×)	1 (0.7×)	2 (1.0×)	3 (4×)	61 (26×)
Q2(a)	1	4	1 (0.7×)	3 (0.8×)	1 (0.8×)	4 (0.9×)	5 (4×)	4 (1×)
Q2(b)	8	36	5 (0.6×)	25 (0.7×)	1 (0.2×)	31 (0.9×)	9 (1.1×)	43 (1.2×)
Q2(c)	25	472	23 (0.9×)	360 (0.8×)	3 (0.1×)	432 (0.9×)	25 (1×)	451 (1×)
Q2(d)	32	18	30 (0.9×)	19 (1.0×)	6 (0.2×)	19 (1.1×)	118 (4×)	57 (3×)
Q3	13	45	9 (0.7×)	43 (0.9×)	1 (0.1×)	46 (1.0×)	158 (13×)	313 (7×)
Q4	26	N/A	25 (0.9×)	N/A	16 (0.6×)	N/A	103 (4×)	N/A
Q5	1	4	1 (0.8×)	3 (0.6×)	1 (0.4×)	4 (0.9×)	24 (19×)	13 (3×)
Q6(a)	2	14	2 (0.9×)	13 (0.9×)	1 (0.4×)	15 (1.1×)	29 (16×)	19 (1.4×)
Q6(b)	12	11	11 (0.9×)	8 (0.7×)	2 (0.2×)	11 (0.9×)	53 (5×)	66 (6×)



(a) Microbenchmarks



(b) Composite & virtual reality benchmarks

Figure 5.6: VDBMS performance showing, at various scale factors, the total time to execute the Visual Road benchmark queries.

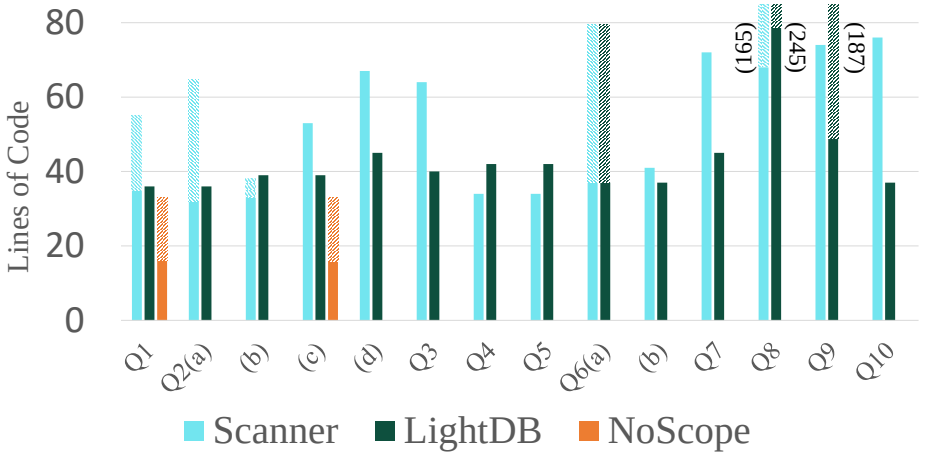


Figure 5.7: Lines of code (LOC) required to execute each Visual Road benchmark query. Solid bars show LOC to implement query; hashed bars show LOC for supporting extension code. Values over 80 LOC shown in parenthesis. NoScope can only express Q1 and Q2(c), and so we omit its bars for other queries.

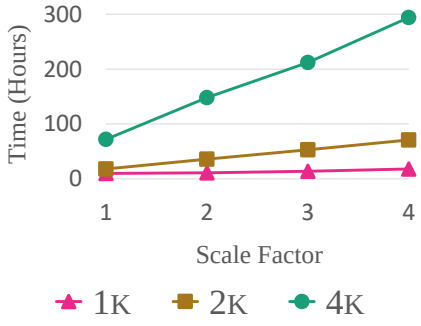


Figure 5.8: Performance by scale/resolution (4 nodes, 60 min)

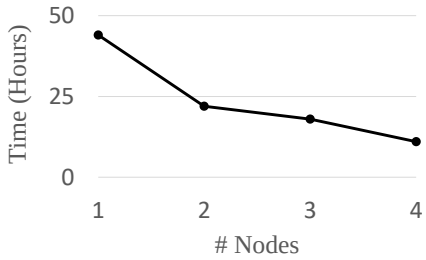


Figure 5.9: Performance by #nodes (scale 2, 1k, 60 min).

Chapter 6

RELATED WORK

In this chapter we review prior work related to the three core chapters of this thesis. First, Section 6.1 describes related video data management systems, video analytics systems, virtual and augmented reality frameworks, and data models. Next, Section 6.2 discusses work related to file system support of video data (e.g., caching, edge storage, and related abstractions). Finally, in Section 6.3 describes various data management benchmarks, along with video-oriented datasets and simulation frameworks used to evaluate the performance of video and related data management systems.

6.1 Video Database Management Systems

A substantial body of work exists that relates to the management of video data, including the recent systems categorized in Table 6.1. Most of these systems [94, 98, 122, 3, 169, 89, 57, 124] target general purpose 2D video analytics and processing. VideoStorm [169] allow users to express distributed analytical workloads over 2D video (e.g., citywide security feeds), while Optasia [98] also supports declarative queries. Similarly, VAIT [94] offers a small set of predefined queries over large video datasets. However, none of these systems offer a high-level way in order to reason about multiple video streams, nor do they support reasoning about camera position, orientation, or overlap. Additionally, since these systems are limited to 2D video analytics, they do not readily support virtual reality or augmented reality video, and force developers to manually map 3D environments onto 2D constructs. These factors collectively result in rigid applications that are difficult to maintain and evolve. Other 2D

Table 6.1: Video data systems & frameworks. Bolded systems have source available and can execute the applications described in Chapter 3 (see Section 3.2.4).

Type	Systems
2D General Purpose	**FFmpeg** [17], **GStreamer** [57], **GPAC** [49] (also supports 360°)
2D Vision	**OpenCV** [118], **OpenIMAG** [61], Panorama [171]
2D Analytics	**Scanner** [124], BlazeIt [85], Optasia [98], VideoStorm [169], Rocket [3]
2D Content/Feature Search	VDBMS [10], BilVideo [41], VideoQ [25], AVIS [1], DelaunayMM [35]
2D Metadata Search	VideoAnywhere [139], OVID [116]
2D Presentation	MINOS [28]
Streaming & Transcoding	Morph [52], 360ProbDASH [162]

video-oriented systems variously provide content-based [121, 28, 116, 85, 25], keyword or metadata [139, 35], and similarity or feature-based search [115, 24, 41, 10, 61] for 2D video and images.

More specialized VR and AR systems explore efficient video delivery that targets a single format or workload. Examples include dedicated 360° streaming systems [96, 162] and light field image-based rendering systems [14, 127]. While many of these systems support viewer position or orientation, like their 2D counterparts they do not support the ability to reason about cameras or the other characteristics highlighted in Chapter 1.

LightDB's temporal light field (TLF) data model and algebra supports much richer workloads than these systems. For example, the GStreamer [57] and Scanner [124] systems allow for fixed pipelines that are similar to composed TLF algebraic operations (and LightDB query plans), but these pipelines are rigid, closely tied to a physical execution strategy, and also require manually mapping constructs such as camera orientation and projection onto a 2D equivalent. Finally, the SQL multimedia (SQL/MM) standard [142] extends the SQL specification to a limited set of operations (e.g., cropping, color histograms) over 2D images, but does not support video, multiple cameras, user-defined extensions.

The database community has explored the application of array-based data models that are similar to the TLF model exposed by LightDB. For example, the RasDaMan [15], SciDB [22], and TileDB [120] DBMSs allow developers to define and operate over multidimensional arrays. While these systems offer excellent performance for scientific and other analytical workloads, they do not take advantage of the unique nature of visual world applications (VWAs) such as support for multiple cameras, orientation, virtual and augmented reality video, continuousness, angular periodicity, and nonuniform projections. Additionally, existing array DBMSs do not support video compression and its idiosyncrasies. As we show in our evaluation of LightDB, first-class support for these constructs leads to dramatically improved performance.

Other work (e.g., [54, 105, 107]) focuses on capture, stitching, and depth estimation aspects of 360° video. Similar examples exist for light field capture [90]. These efforts are complementary to the LightDB system presented in Chapter 3, which accepts preprocessed 360° and light field videos from these pipelines and performs further query processing. They, however, do not generalize to mixtures of 2D and 3D video and do not allow for the expression of arbitrary VWAs.

In the evaluation of LightDB (see Section 3.5), we demonstrated a substantial reduction in total data transfer by tiling a 360° video sphere and adaptively delivering tiles at various qualities. Recent work has shown similar performance improvements [67, 108, 48, 56, 106]. These applications and approaches, however, are dedicated exclusively to the task of 2D or 360° video tiling and do not generalize to other VR and AR workloads. Birklbauer et al. [20] show similar advantages for light field rendering.

6.2 File System Support for Video Data

As highlighted in the previous section, increased interest in applied machine learning and computer vision has led to the development of a number of new systems that target video analytics. However, these systems continue to read and write video to a local or distributed file system as opaque, coarse-grained entities and thereby suffer from the shortcomings discussed

in Chapter 1 (in particular those described in the visual world application impedance mismatch; see Definition 1.1). Video-oriented deep learning accelerators such as BlazeIt [85], VideoStorm [169], Focus [73], and NoScope [86] suffer from similar shortcomings. Collectively, these systems can transparently benefit from the approach described in Chapter 4 and implemented in VFS.

While the systems community has a long history of introducing specialized file systems (e.g., HDFS [140]), few prior systems have targeted video analytics (although other authors have identified the need for systems like VFS [58, 84]). VStore [164] is one such example that targets machine learning workloads by staging video in a pre-specified set of formats. However, VStore requires that the developer to know beforehand the specific workload being optimized, lacks the ability to efficiently read and write beyond these workload formats, and does not take advantage of data independence to improve performance beyond persisting these fixed materializations. Similarly, quFiles [152] offers a file system abstraction but does not exploit video data independence in a granular manner (i.e., at the GOP or frame level). Other authors have looked at optimizing on-disk layout of video data in the context of scalable streaming [87]. Finally, related storage-oriented systems such as Haystack [16] and VDMS [130] emphasize image-based operations and metadata access.

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) [76, 81, 4, 163, 23]. Since these cameras have constrained storage and compute resources, they would benefit from a storage system such as VFS that can transparently balance these factors and improve performance.

Finally, the database community has a long history of exploiting data independence in order to improve performance, which is a key technique used by VFS to obtain its performance advantages. For example, it transparently employs Zstandard [46] compression rather than a typical video codec. Other orthogonal optimizations could be employed to further improve VFS's performance such as Vignette [108], or the homomorphic operators described in LightDB (see Chapter 3) [65].

6.3 Video Performance Benchmarking

The database community has a long history of standardizing on various benchmarks, which target a wide range of data models. These include longstanding areas such as OLTP [149], and OLAP [124], and streaming [8]. They also cover more modern areas such as the Internet of things [11], block chains [40], social networks [44], and big data analytics [148].

However, we are aware of no video performance benchmark that scales to an unlimited duration or resolution and does not require manual annotation, despite the fact that a number of recent video database management systems (VDBMSs) have emerged to support a wide range of modern applications (see Table 6.1). To fill this gap, the Visual Road benchmark described in Chapter 5 complements existing data management benchmarks by extending robust support for performance evaluation in the domain of video processing at scale and motivates subsequent work in improving video querying functionality and performance. To do this, Visual Road's query suite (see Section 5.3) targets functionality beyond simple content search and information retrieval, including license plate recognition [98, 169] (Q8), background subtraction/masking [98] (Q2(d), Q6), general object detection [65, 124, 83, 85, 73, 169] (Q2(c), Q7), decode performance [65, 124], stitching [124, 74] (Q9), up/downsampling [74] (Q4,Q5,Q10), user-defined transformations [6] (Q2(a-d)), and tile-based encoding [65, 72, 168, 67] (Q10). Each of the systems cited in Table 6.1 benefits from a robust benchmark such as Visual Road, which allows for an objective comparison of features and performance.

A number of video-oriented datasets and benchmarks have emerged that target various aspects of machine learning. For example, the UA-DETRAC [159] suite targets multi-object detection and tracking. Other datasets such as BDD100K [167], ApolloScape [75], and the Waymo Open Dataset [158] target autonomous driving. Still other datasets target diverse areas such as traffic density estimation (e.g., WebCamT [170]) or human activity (e.g., ActivityNet [68]). While these datasets might be useful in evaluating VDBMS performance, they are of fixed, modest size and must be laboriously annotated.

Similarly, due to recent advances in autonomous driving, there have been a number of recent video-oriented simulation frameworks released that relate to this specific aspect of AI model training. Related simulation-oriented frameworks include CARLA [42]—on which the Visual Road prototype is built—along with others such as AirSim [138] and DeepDrive [39]. Other areas have seen similar advances, such as in military (e.g., UTSAF [125]) and medical applications (e.g., [33]). While these frameworks all use modern simulation and visualization software (e.g., Unreal), they are not designed to produce large amounts of heterogeneous video, nor do they come with a query suite useful for evaluating VDBMS performance.

Finally, generalizability and transferability of results is a significant challenge to applications that leverage synthetic data for use in real-world applications. Prior work in several areas have examined this issue. For example, in their survey of robot simulators, Craighead et al. argued that contemporaneous simulation software had high physical fidelity [34]. In a subsequent survey on UAV and robot simulators, Cook et al. drew similar conclusions in oceanographic robotics with respect to the physics engines (i.e., accuracy of rigid body dynamics, collision detection) [31]. In the computer vision domain, researchers have evaluated the transferability of models learned on synthetic data to real-world applications. Previous approaches have been variously quantitative (e.g., precision/recall [62], multi-object tracking accuracy [51], collision-free percentage [132], average accuracy [38], ROC curve [104]) or qualitative (e.g., observed similarity [138]). Some previous work has demonstrated that synthetic data leads to superior models when data is limited or of low variety [62]. Visual Road evaluates transferability using an approach similar to that described by Hattori et al. [62].

Chapter 7

CONCLUSIONS & FUTURE DIRECTIONS

Our world is now filled with vast networks of correlated cameras which capture what is happening around us from many diverse perspectives and positions. Today's video data management systems (VDBMSs), however, continue to assume that video streams are independent and two dimensional. They lack data models that represent the real world and require developers to manually track details such as camera and viewer position and orientation. They make developers responsible for combining, aligning, downsampling, overlaying, and intermixing sets of interrelated videos. They treat video data as if it were independent, isolated, and stored on disk as opaque blobs. Collectively, these factors lead to the impedance mismatch defined in Chapter 1 (see Definition 1.1).

In this thesis we presented systems designed to remedy these issues. In Chapter 3 we introduced LightDB, a VDBMS designed to efficiently process virtual and augmented (VR and AR) video. LightDB exposes a data model that treats such video data as a logically continuous six-dimensional light field. It offers a query language and algebra, allowing for efficient declarative queries. Our LightDB prototype show that queries in LightDB are easily expressible and yield up to a 500× performance improvement relative to other video processing frameworks.

LightDB relies on a new video file system (VFS; see Chapter 4) which improves the performance of video-oriented applications and data management systems. VFS decouples high-level operations such as computer vision and machine learning algorithms from the low-level plumbing required to read and write data in a suitable format. Users leverage VFS by reading and writing video data in whatever format is most useful, and VFS transparently identifies the most efficient method to retrieve that video data. Our experiments showed that,

relative to both local storage and other dedicated video storage systems, VFS offers more flexible read and write formats and reduces read time by up to 54%. Our optimizations also decrease the cost of persisting video by up to 45%.

Finally, in Chapter 5 we presented Visual Road, a benchmark for evaluating VDBMS performance. Visual Road comes with a data generator that produces an unlimited amount of synthetic video generated by simulating an active metropolitan area, along with a suite of queries that evaluate VDBMS performance. Our results show that video generated using Visual Road closely matches real-world, manually-annotated video corpora and allows VDBMSs to be evaluated at any scale.

Collectively, these systems free developers from the impedance mismatch described in Chapter 1 (see Definition 1.1), where developers need not be concerned about the low-level details of video persistence and structure, and where domain-appropriate optimizations, such as targeted deep-learning compression, are applied automatically without manual intervention.

Beyond these contributions, additional questions remain. For example, consider multiple traffic and vehicle-mounted cameras that capture the scene of an automobile accident. While each camera captures one perspective and potentially the scene of the accident, combined camera output can help better reconstruct the sequence of events, especially since we can exploit the fact that these video streams come with geospatial information that lets us reason about their position and (potentially overlapping) fields of view.

These multi-camera video "worlds" represent a new, important, and increasingly common way of reasoning about video. An important future research direction involves building a new type of VDBMS around these concepts. Such a system, which we have initially explored in a system called VisualWorldDB [64], represents an initial design and vision for a VDBMS that is optimized for multi-camera video worlds. This system, *built on top* of LightDB [65] and VFS [63] (see Figure 1.2), ingests spatial video data from diverse sources, makes it queryable as one multidimensional visual world, and lets users reason *directly about objects within this world*. For example, a user might query which vehicles left the scene of an accident and where they went, rather than being required to reason about the underlying pixel data.

Final remarks: This thesis introduces systems, techniques, and approaches that improve the storage, management, and analysis of video data. In it we leverage state-of-the-art methods in video compression, signal processing, computer vision, and data storage to explore how we can best integrate these methods with modern data management techniques to improve video application performance and programmability. In sum, we show that application of fundamental data management techniques is critical in addressing the many challenges associated with video data processing, storage, and evaluation.

Bibliography

[1] Sibel Adali, K Selcuk Candan, Kutluhan Erol, and VS Subrahmanian. Avis: An advanced video information system. Technical Report 97-44, 1997.

[2] Edward H. Adelson and James R. Bergen. The plenoptic function and the elements of early vision. In *Computational Models of Visual Processing*, pages 3–20. MIT Press, 1991.

[3] Ganesh Ananthanarayanan, Paramvir Bahl, Peter Bodík, Krishna Chintalapudi, Matthai Philipose, Lenin Ravindranath, and Sudipta Sinha. Real-time video analytics: The killer app for edge computing. *IEEE Computer*, 50(10):58–67, 2017.

[4] Ganesh Ananthanarayanan, Victor Bahl, Landon P. Cox, Alex Crown, Shadi Nogbahi, and Yuanchao Shu. Video analytics - killer app for edge computing. In *MobiSys*, pages 695–696, 2019.

[5] Robert Anderson, David Gallup, Jonathan T. Barron, Janne Kontkanen, Noah Snavely, Carlos Hernández, Sameer Agarwal, and Steven M. Seitz. Jump: virtual reality video. *TOG*, 35(6):198:1–198:13, 2016.

[6] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A serverless video processing framework. In *SoCC*, pages 263–274, 2018.

[7] Apple, Inc. HTTP Live Streaming. RFC 8216, August 2017.

[8] Arvind Arasu, Mitch Cherniack, Eduardo F. Galvez, David Maier, Anurag Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: A stream data management benchmark. In *PVLDB*, pages 480–491, 2004.

[9] Gonzalo R. Arce. *Nonlinear Signal Processing - A Statistical Approach*. Wiley, 2004.

[10] Walid G. Aref, Ann Christine Catlin, Jianping Fan, Ahmed K. Elmagarmid, Moustafa A. Hammad, Ihab F. Ilyas, Mirette S. Marzouk, and Xingquan Zhu. A video database management system for advancing video database research. In *MIS*, pages 8–17, 2002.

[11] Martin F. Arlitt, Manish Marwah, Gowtham Bellala, Amip Shah, Jeff Healey, and Ben Vandiver. Iotabench: an internet of things analytics benchmark. In *ICPE*, pages 133–144, 2015.

[12] Mikhail J. Atallah, Danny Z. Chen, and DT Lee. An optimal algorithm for shortest paths on weighted interval and circular-arc graphs, with applications. *Algorithmica*, 14(5):429–441, 1995.

[13] Emmanouil N Barmpounakis, Eleni I Vlahogianni, and John C Golias. Unmanned aerial aircraft systems for transportation engineering: Current practice and future challenges. *IJTST*, 5(3):111–122, 2016.

[14] Ingo Bauermann, Yang Peng, and Eckehard G. Steinbach. RDTC optimized streaming for remote browsing in image-based scene representations. In *3DPVT*, pages 758–765, 2006.

[15] Peter Baumann, Andreas Dehmel, Paula Furtado, Roland Ritsch, and Norbert Widmann. The multidimensional database system rasdaman. In *SIGMOD*, pages 575–577, 1998.

[16] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in haystack: Facebook's photo storage. In *OSDI*, pages 47–60, 2010.

[17] Fabrice Bellard. FFmpeg. `https://ffmpeg.org`, 2018.

[18] Daniel S. Berger, Nathan Beckmann, and Mor Harchol-Balter. Practical bounds on optimal caching with variable object sizes. *POMACS*, 2(2):32:1–32:38, 2018.

[19] Neil Birkbeck, Chip Brown, and Robert Suderman. Quantitative evaluation of omnidirectional video quality. In *QoMEX*, pages 1–3, 2017.

[20] Clemens Birklbauer, Simon Opelt, and Oliver Bimber. Rendering gigaray light fields. *Computer Graphics Forum*, 32(2):469–478, 2013.

[21] Chip Brown. Bringing pixels front and center in VR video. `https://www.blog.google/products/google-vr/bringing-pixels-front-and-center-vr-video`, 2017.

[22] Paul G. Brown. Overview of SciDB: large scale array storage, processing and analysis. In *SIGMOD*, pages 963–968, 2010.

[23] Jiashen Cao, Ramyad Hadidi, Joy Arulraj, and Hyesoon Kim. Video analytics from edge to server: work-in-progress. In *CODES+ISSS*, pages 14:1–14:2, 2019.

[24] Chad Carson, Megan Thomas, Serge J. Belongie, Joseph M. Hellerstein, and Jitendra Malik. Blobworld: A system for region-based image indexing and retrieval. In *VISUAL*, pages 509–516, 1999.

[25] Shih-Fu Chang, William Chen, Horace J. Meng, Hari Sundaram, and Di Zhong. Videoq: An automated content based video search system using visual cues. In *MMSys*, pages 313–324, 1997.

[26] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. Glimpse: Continuous, real-time object recognition on mobile devices. In *SenSys*, pages 155–168, 2015.

[27] Aakanksha Chowdhery and Mung Chiang. Model predictive compression for drone video analytics. In *SECON*, pages 19–23, 2018.

[28] Stavros Christodoulakis, F. Ho, and M. Theodoridou. The multimedia object presentation manager of MINOS: A symmetric approach. In *SIGMOD*, pages 295–310, 1986.

[29] Cloudview. Visual IoT: Where the IoT cloud and big data come together. 2018.

[30] Contributors to the FFmpeg Community Documentation Wiki. FFmpeg: Concatenating media files. `https://trac.ffmpeg.org/wiki/Concatenate`.

[31] Daniel Cook, Andrew Vardy, and Ron Lewis. A survey of auv and robot simulators for multi-vehicle operations. In *AUV*, pages 1–8, 2014.

[32] Xavier Corbillon, Francesca De Simone, and Gwendal Simon. 360-degree video head movement dataset. In *MMSys*, pages 199–204, 2017.

[33] Brent Cowan, Hamed Sabri, Bill Kapralos, Sayra Cristancho, Fuad Moussa, and Adam Dubrowski. SCETF: serious game surgical cognitive education and training framework. In *IGIC*, pages 130–133, 2011.

[34] Jeff Craighead, Robin R. Murphy, Jenny Burke, and Brian F. Goldiez. A survey of commercial & open source unmanned vehicle simulators. In *ICRA*, pages 852–857, 2007.

[35] Isabel F. Cruz and Wendy T. Lucas. Delaunay$^{\mathrm{mm}}$: A visual framework for multimedia presentation. In *VL*, pages 216–223, 1997.

[36] Maureen Daum, Brandon Haynes, Dong He, Amrita Mazumdar, Magdalena Balazinska, and Alvin Cheung. TASM: A tile-based storage manager for video analytics. *CoRR*, abs/2006.02958, 2020.

[37] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[38] César Roberto de Souza, Adrien Gaidon, Yohann Cabon, and Antonio Manuel López Peña. Procedural generation of videos to train deep action recognition networks. In *CVPR*, pages 2594–2604, 2017.

[39] DeepDrive: Self-driving ai. `https://deepdrive.io`, 2018.

[40] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. BLOCKBENCH: A framework for analyzing private blockchains. In *SIGMOD*, pages 1085–1100, 2017.

[41] Mehmet Emin Dönderler, Ediz Saykol, Umut Arslan, Özgür Ulusoy, and Ugur Güdükbay. Bilvideo: Design and implementation of a video database management system. *MTAP*, 27(1):79–104, 2005.

[42] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In *CoRL*, pages 1–16, 2017.

[43] Epic Games. Unreal engine 4. `https://www.unrealengine.com`, 2019.

[44] Orri Erling, Alex Averbuch, Josep-Lluís Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat-Pérez, Minh-Duc Pham, and Peter A. Boncz. The LDBC social network benchmark: Interactive workload. In *SIGMOD*, pages 619–630, 2015.

[45] Mark Everingham, Luc J. Van Gool, Christopher K. I. Williams, John M. Winn, and Andrew Zisserman. The pascal visual object classes (VOC) challenge. *IJCV*, 88(2):303–338, 2010.

[46] Facebook. Zstandard real-time compression algorithm. `https://facebook.github.io/zstd`.

[47] Facebook Surround 360. `https://facebook360.fb.com/facebook-surround-360`.

[48] Jean Le Feuvre and Cyril Concolato. Tiled-based adaptive streaming using MPEG-DASH. In *MMSys*, pages 41:1–41:3, 2016.

[49] Jean Le Feuvre, Cyril Concolato, and Jean-Claude Moissinac. GPAC: open source multimedia framework. In *ICME*, pages 1009–1012, 2007.

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

[51] Adrien Gaidon, Qiao Wang, Yohann Cabon, and Eleonora Vig. Virtualworlds as proxy for multi-object tracking analysis. In *CVPR*, pages 4340–4349, 2016.

[52] Guanyu Gao and Yonggang Wen. Morph: A fast and scalable cloud transcoding system. In *MM*, pages 1160–1163, 2016.

[53] Shilpa George, Junjue Wang, Mihir Bala, Thomas Eiszler, Padmanabhan Pillai, and Mahadev Satyanarayanan. Towards drone-sourced live video analytics for the construction industry. In *HotMobile*, pages 3–8, 2019.

[54] Google. Google Jump. `https://www.google.com/get/cardboard/jump`.

[55] Google Poly. `https://poly.google.com`.

[56] Mario Graf, Christian Timmerer, and Christopher Müller. Towards bandwidth efficient adaptive streaming of omnidirectional video over HTTP: design, implementation, and evaluation. In *MMSys*, pages 261–271, 2017.

[57] GStreamer Team. Gstreamer: open source multimedia framework. `https://gstreamer.freedesktop.org`.

[58] Vishakha Gupta-Cledat, Luis Remis, and Christina R. Strong. Addressing the dark side of vision research: Storage. In *HotStorage*, 2017.

[59] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

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

[61] Jonathon S. Hare, Sina Samangooei, and David Dupplaw. Openimaj and imageterrier: Java libraries and tools for scalable multimedia analysis and indexing of images. In *ICME*, pages 691–694, 2011.

[62] Hironori Hattori, Vishnu Naresh Boddeti, Kris M. Kitani, and Takeo Kanade. Learning scene-specific pedestrian detectors without real data. In *CVPR*, pages 3819–3827, 2015.

[63] Brandon Haynes, Maureen Daum, Amrita Mazumdar, Magdalena Balazinska, Luis Ceze, and Alvin Cheung. VFS: A file system for video analytics. *Under review*, 2020.

[64] Brandon Haynes, Maureen Daum, Amrita Mazumdar, Magdalena Balazinska, Alvin Cheung, and Luis Ceze. VisualWorldDB: A DBMS for the visual world. In *CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*, 2020.

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

[66] Brandon Haynes, Amrita Mazumdar, Magdalena Balazinska, Luis Ceze, and Alvin Cheung. Visual Road: A video data management benchmark. In *SIGMOD*, pages 972–987, 2019.

[67] Brandon Haynes, Artem Minyaylov, Magdalena Balazinska, Luis Ceze, and Alvin Cheung. VisualCloud demonstration: A DBMS for virtual reality. In *SIGMOD*, pages 1615–1618, 2017.

[68] Fabian Caba Heilbron, Victor Escorcia, Bernard Ghanem, and Juan Carlos Niebles. Activitynet: A large-scale video benchmark for human activity understanding. In *CVPR*, pages 961–970, 2015.

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

[70] Simon Heinzle, Pierre Greisen, David Gallup, Christine Chen, Daniel Saner, Aljoscha Smolic, Andreas Burg, Wojciech Matusik, and Markus H. Gross. Computational stereo camera system with programmable control loop. *TOG*, 30(4):94:1–94:10, 2011.

[71] Alain Horé and Djemel Ziou. Image quality metrics: PSNR vs. SSIM. In *ICPR*, pages 2366–2369, 2010.

[72] Mohammad Hosseini and Viswanathan Swaminathan. Adaptive 360 VR video streaming based on MPEG-DASH SRD. In *ISM*, pages 407–408, 2016.

[73] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodík, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. Focus: Querying large video datasets with low latency and low cost. In *OSDI*, pages 269–286, 2018.

[74] Qi Huang, Petchean Ang, Peter Knowles, Tomasz Nykiel, Iaroslav Tverdokhlib, Amit Yajurvedi, Paul Dapolito IV, Xifan Yan, Maxim Bykov, Chuen Liang, Mohit Talwar, Abhishek Mathur, Sachin Kulkarni, Matthew Burke, and Wyatt Lloyd. SVE: distributed video processing at facebook scale. In *SOSP*, pages 87–103, 2017.

[75] Xinyu Huang, Xinjing Cheng, Qichuan Geng, Binbin Cao, Dingfu Zhou, Peng Wang, Yuanqing Lin, and Ruigang Yang. The apolloscape dataset for autonomous driving. *CoRR*, abs/1803.06184, 2018.

[76] Chien-Chun Hung, Ganesh Ananthanarayanan, Peter Bodík, Leana Golubchik, Minlan Yu, Paramvir Bahl, and Matthai Philipose. Videoedge: Processing camera streams using hierarchical clusters. In *SEC*, pages 115–131, 2018.

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

[78] Sinisa Ilic, Mile Petrovic, Branimir Jaksic, Petar Spalevic, Ljubomir Lazic, and Mirko Milosevic. Experimental analysis of picture quality after compression by different methods. *Przeglad Elektrotechniczny*, pages 0033–2097, 2013.

[79] International Organization for Standardization. Coding of audio-visual objects – part 14: MP4 file format. Standard ISO/IEC 14496-14:2003, 2003.

[80] International Organization for Standardization. Information technology—Dynamic adaptive streaming over HTTP (DASH)—Part 1: Media presentation description and segment formats. Standard ISO/IEC 23009-1:2014, 2014.

[81] Samvit Jain, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, and Joseph Gonzalez. Scaling video analytics systems to large camera deployments. In *HotMobile*, pages 9–14, 2019.

[82] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *ACMMM*, pages 675–678, 2014.

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

[84] Junchen Jiang, Yuhao Zhou, Ganesh Ananthanarayanan, Yuanchao Shu, and Andrew A. Chien. Networked cameras are the new big data clusters. HotEdgeVideo'19, pages 1–7, 2019.

[85] Daniel Kang, Peter Bailis, and Matei Zaharia. Blazeit: Optimizing declarative aggregation and limit queries for neural network-based video analytics. *PVLDB*, 13(4):533–546, 2019.

[86] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Noscope: Optimizing deep CNN-based queries over video streams at scale. *PVLDB*, 10(11):1586–1597, 2017.

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

[88] Changil Kim, Alexander Hornung, Simon Heinzle, Wojciech Matusik, and Markus H. Gross. Multi-perspective stereoscopy from light fields. *ACM Trans. Graph.*, 30(6):190:1–190:10, 2011.

[89] Sanjay Krishnan, Adam Dziedzic, and Aaron J. Elmore. Deeplens: Towards a visual data management system. In *CIDR*, 2019.

[90] Bernd Krolla, Maximilian Diebold, Bastian Goldlücke, and Didier Stricker. Spherical light fields. In *BMVC*, 2014.

[91] Marc Levoy and Pat Hanrahan. Light field rendering. In *SIGGRAPH*, pages 31–42, 1996.

[92] The reference implementation of the linux FUSE (filesystem in userspace) interface. `https://github.com/libfuse/libfuse`.

[93] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. In *ECCV*, pages 740–755, 2014.

[94] Siyuan Liu, Jiansu Pu, Qiong Luo, Huamin Qu, Lionel M. Ni, and Ramayya Krishnan. VAIT: A visual analytics system for metropolitan transportation. *ITS*, 14(4):1586–1596, 2013.

[95] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: single shot multibox detector. In *ECCV*, pages 21–37, 2016.

[96] Xing Liu, Qingyang Xiao, Vijay Gopalakrishnan, Bo Han, Feng Qian, and Matteo Varvello. 360° innovations for panoramic video streaming. In *HotNets*, pages 50–56, 2017.

[97] David G. Lowe. Object recognition from local scale-invariant features. In *ICCV*, pages 1150–1157, 1999.

[98] Yao Lu, Aakanksha Chowdhery, and Srikanth Kandula. Optasia: A relational platform for efficient large-scale video analytics. In *SoCC*, pages 57–70, 2016.

[99] Yao Lu, Aakanksha Chowdhery, Srikanth Kandula, and Surajit Chaudhuri. Accelerating machine learning inference with probabilistic predicates. In *SIGMOD*, pages 1493–1508, 2018.

[100] Lytro. `https://vr.lytro.com`.

[101] Lytro Immerge. `https://www.lytro.com/press/releases/lytro-brings-revolutionary-light-field-technology-to-film-\and-tv-production-with-lytro-cinema`.

[102] Will Maddern, Geoff Pascoe, Chris Linegar, and Paul Newman. 1 Year, 1000km: The Oxford RobotCar Dataset. *IJRR*, 36(1):3–15, 2017.

[103] Magic Leap. `https://www.magicleap.com`.

[104] Javier Marín, David Vázquez, David Gerónimo, and Antonio M. López. Learning appearance in virtual scenarios for pedestrian detection. In *CVPR*, pages 137–144, 2010.

[105] Kevin Matzen, Michael F. Cohen, Bryce Evans, Johannes Kopf, and Richard Szeliski. Low-cost 360 stereo photography and video capture. *TOG*, 36(4):148:1–148:12, 2017.

[106] Aditya Mavlankar and Bernd Girod. Pre-fetching based on video analysis for interactive region-of-interest streaming of soccer sequences. In *ICIP*, pages 3061–3064, 2009.

[107] Amrita Mazumdar, Armin Alaghi, Jonathan T. Barron, David Gallup, Luis Ceze, Mark Oskin, and Steven M. Seitz. A hardware-friendly bilateral solver for real-time virtual reality video. In *HPG*, pages 13:1–13:10, 2017.

[108] Amrita Mazumdar, Brandon Haynes, Magda Balazinska, Luis Ceze, Alvin Cheung, and Mark Oskin. Perceptual compression for video storage and processing systems. In *SoCC*, pages 179–192, 2019.

[109] Tim Milliron, Chrissy Szczupak, and Orin Green. Hallelujah: The world's first Lytro VR experience. In *SIGGRAPH*, pages 7:1–7:2, 2017.

[110] Kiran M. Misra, C. Andrew Segall, Michael Horowitz, Shilin Xu, Arild Fuldseth, and Minhua Zhou. An overview of tiles in HEVC. *Journal of Selected Topics in Signal Processing*, 7(6):969–977, 2013.

[111] WebVTT parsing library. `https://github.com/hihihippp/webvtt-3`, 2018.

[112] Richard A. Newcombe, Dieter Fox, and Steven M. Seitz. Dynamicfusion: Reconstruction and tracking of non-rigid scenes in real-time. In *CVPR*, pages 343–352, 2015.

[113] Nvidia video codec. `https://developer.nvidia.com/nvidia-video-codec-sdk`.

[114] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. NVIDIA Corporation, 2007.

[115] Virginia E. Ogle and Michael Stonebraker. Chabot: Retrieval from a relational database of images. *IEEE Computer*, 28(9):40–48, 1995.

[116] Eitetsu Oomoto and Katsumi Tanaka. OVID: design and implementation of a video-object database system. *TKDE*, 5(4):629–643, 1993.

[117] OpenALPR Technology. Open automatic license plate recognition library. `http://www.openalpr.com`, 2018.

[118] OpenCV. Open Source Computer Vision Library. `https://opencv.org`, 2018.

[119] Oracle. Oracle multimedia: Managing multimedia content. Technical report, 2009.

[120] Stavros Papadopoulos, Kushal Datta, Samuel Madden, and Timothy G. Mattson. The TileDB array data storage manager. *PVLDB*, 10(4):349–360, 2016.

[121] B. V. Patel and B. B. Meshram. Content based video retrieval systems. *CoRR*, abs/1205.1641, 2012.

[122] Yao Peng, Hao Ye, Yining Lin, Yixin Bao, Zhijian Zhao, Haonan Qiu, Yao Lu, Li Wang, and Yingbin Zheng. Large-scale video classification with elastic streaming sequential data processing system. In *LSVC*, 2017.

[123] Silvia Pfeiffer. WebVTT: The web video text tracks format. Candidate recommendation, W3C, May 2018. https://www.w3.org/TR/2018/CR-webvtt1-20180510/.

[124] Alex Poms, Will Crichton, Pat Hanrahan, and Kayvon Fatahalian. Scanner: efficient video analysis at scale. *TOG*, 37(4):138:1–138:13, 2018.

[125] Phongsak Prasithsangaree, Joseph Manojlovich, Jinlin Chen, and Michael Lewis. UT-SAF: a simulation bridge between onesaf and the unreal game engine. In *SMC*, pages 1333–1338, 2003.

[126] Datta Krupa R., Aniket Basu Roy, Minati De, and Sathish Govindarajan. Demand hitting and covering of intervals. In *ACALDAM*, pages 267–280, 2017.

[127] Prashant Ramanathan, Mark Kalman, and Bernd Girod. Rate-distortion optimized interactive light field streaming. *IEEE Transactions on Multimedia*, 9(4):813–825, 2007.

[128] Xukan Ran, Haoliang Chen, Zhenming Liu, and Jiasi Chen. Delivering deep learning to mobile devices via offloading. In *Network@SIGCOMM*, pages 42–47, 2017.

[129] Joseph Redmon and Ali Farhadi. YOLO9000: better, faster, stronger. In *CVPR*, pages 6517–6525, 2017.

[130] Luis Remis, Vishakha Gupta-Cledat, Christina R. Strong, and Ragaad Altarawneh. VDMS: an efficient big-visual-data access for machine learning workloads. *CoRR*, abs/1810.11832, 2018.

[131] J. Rogers et al. Overview of SciDB: Large scale array storage, processing and analysis. In *SIGMOD*, 2010.

[132] Fereshteh Sadeghi and Sergey Levine. CAD2RL: real single-image flight without a single real image. In *RSS*, 2017.

[133] David Salomon. *The Computer Graphics Manual*. Texts in Computer Science. Springer, 2011.

[134] Daniel Scharstein and Richard Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International Journal of Computer Vision*, 47(1-3):7–42, 2002.

[135] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. Rtp: A transport protocol for real-time applications. RFC 3550, July 2003.

[136] Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. Overview of the scalable video coding extension of the H.264/AVC standard. *TCSVT*, 17(9):1103–1120, 2007.

[137] Michael Seufert, Sebastian Egger, Martin Slanina, Thomas Zinner, Tobias Hoßfeld, and Phuoc Tran-Gia. A survey on quality of experience of HTTP adaptive streaming. *IEEE Communications Surveys and Tutorials*, 17(1):469–492, 2015.

[138] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *FSR*, pages 621–635, 2017.

[139] Amit P. Sheth, Clemens Bertram, and Kshitij Shah. Videoanywhere: A system for searching and managing distributed video assets. *SIGMOD Record*, 28(1):104–109, 1999.

[140] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *MSST*, pages 1–10, 2010.

[141] Spherical video V2 RFC. `https://github.com/google/spatial-media/blob/master/docs/spherical-video-v2-rfc.md`.

[142] Knut Stolze. SQL/MM spatial - the standard to manage spatial data in a relational database system. In *BTW*, pages 247–264, 2003.

[143] Gary Sullivan and Stephen Estrop. Recommended 8-bit yuv formats for video rendering. https://docs.microsoft.com/en-us/windows/desktop/medfound/recommended-8-bit-yuv-formats-for-video-rendering, 2008.

[144] Gary J. Sullivan, Jens-Rainer Ohm, Woojin Han, and Thomas Wiegand. Overview of the high efficiency video coding (HEVC) standard. *TCSVT*, 22(12):1649–1668, 2012.

[145] Gary J. Sullivan and Thomas Wiegand. Video compression - from concepts to the H.264/AVC standard. *IEEE*, 93(1):18–31, 2005.

[146] Ardi Tampuu, Maksym Semikin, Naveed Muhammad, Dmytro Fishman, and Tambet Matiisen. A survey of end-to-end driving: Architectures and training methods. *CoRR*, abs/2003.06404, 2020.

[147] The WebM Project. The WebM project. https://www.webmproject.org.

[148] Xinhui Tian, Shaopeng Dai, Zhihui Du, Wanling Gao, Rui Ren, Yaodong Cheng, Zhifei Zhang, Zhen Jia, Peijian Wang, and Jianfeng Zhan. Bigdatabench-s: An open-source scientific big data benchmark suite. In *IPDPS*, pages 1068–1077, 2017.

[149] Transaction Processing Performance Council. TPC-C benchmark. http://www.tpc.org/tpcc, 2018.

[150] Transaction Processing Performance Council. TPC-H benchmark. http://www.tpc.org/tpch, 2018.

[151] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or not to FUSE: performance of user-space file systems. In *USENIX*, pages 59–72, 2017.

[152] Kaushik Veeraraghavan, Jason Flinn, Edmund B. Nightingale, and Brian Noble. qufiles: The right file at the right time. *TOS*, 6(3):12:1–12:28, 2010.

[153] Google VR View. https://developers.google.com/vr/concepts/vrview.

[154] Junjue Wang, Ziqiang Feng, Zhuo Chen, Shilpa George, Mihir Bala, Padmanabhan Pillai, Shao-Wen Yang, and Mahadev Satyanarayanan. Bandwidth-efficient live video analytics for drones via edge computing. In *SEC*, pages 159–173, 2018.

[155] Ting-Chun Wang, Jun-Yan Zhu, Nima Khademi Kalantari, Alexei A. Efros, and Ravi Ramamoorthi. Light field video capture using a learning-based hybrid imaging system. *TOG*, 36(4):133:1–133:13, 2017.

[156] Xiaoli Wang, Aakanksha Chowdhery, and Mung Chiang. Skyeyes: adaptive video streaming from UAVs. In *HotWireless@MobiCom*, pages 2–6, 2016.

[157] Xiaoli Wang, Aakanksha Chowdhery, and Mung Chiang. Networked drone cameras for sports streaming. In *ICDCS*, pages 308–318, 2017.

[158] Waymo open dataset. https://waymo.com/open.

[159] Longyin Wen, Dawei Du, Zhaowei Cai, Zhen Lei, Ming-Ching Chang, Honggang Qi, Jongwoo Lim, Ming-Hsuan Yang, and Siwei Lyu. UA-DETRAC: A new benchmark and protocol for multi-object detection and tracking. *arXiv CoRR*, abs/1511.04136, 2015.

[160] Thomas Wiegand, Gary J. Sullivan, Gisle Bjntegaard, and Ajay Luthra. Overview of the H.264/AVC video coding standard. *TCSVT*, 13(7):560–576, 2003.

[161] Susan Wojcicki. The potential unintended consequences of article 13. https://youtube-creators.googleblog.com/2018/11/i-support-goals-of-article-13-i-also.html, 2018.

[162] Lan Xie, Zhimin Xu, Yixuan Ban, Xinggong Zhang, and Zongming Guo. 360probdash: Improving qoe of 360 video streaming using tile-based HTTP adaptive streaming. In *MMSYS*, pages 315–323, 2017.

[163] Mengwei Xu, Tiantu Xu, Yunxin Liu, Xuanzhe Liu, Gang Huang, and Felix Xiaozhu Lin. Supporting video queries on zero-streaming cameras. *CoRR*, abs/1904.12342, 2019.

[164] Tiantu Xu, Luis Materon Botelho, and Felix Xiaozhu Lin. Vstore: A data store for analytics on large videos. In *EuroSys*, pages 16:1–16:17, 2019.

[165] Billy Yates. Body worn cameras: Making them mandatory. 2018.

[166] YouTube - Virtual Reality. `https://www.youtube.com/vr`.

[167] Fisher Yu, Wenqi Xian, Yingying Chen, Fangchen Liu, Mike Liao, Vashisht Madhavan, and Trevor Darrell. BDD100K: A diverse driving video database with scalable annotation tooling. *CoRR*, abs/1805.04687, 2018.

[168] Alireza Zare, Alireza Aminlou, Miska M. Hannuksela, and Moncef Gabbouj. Hevc-compliant tile-based streaming of panoramic video for virtual reality applications. In *ACMMM*, pages 601–605, 2016.

[169] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodík, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. Live video analytics at scale with approximation and delay-tolerance. In *NSDI*, pages 377–392, 2017.

[170] Shanghang Zhang, Guanhang Wu, João P. Costeira, and José M. F. Moura. Understanding traffic density from large-scale web camera data. In *CVPR*, pages 4264–4273, 2017.

[171] Yuhao Zhang and Arun Kumar. Panorama: A data system for unbounded vocabulary querying over video. *VLDB*, 13(4):477–491, 2019.

[172] Jiazhen Zhou, Rose Qingyang Hu, and Yi Qian. A scalable vehicular network architecture for traffic information sharing. *J-SAC*, 31(9-Supplement):85–93, 2013.