LightDB: A DBMS for Virtual Reality Video

Brandon Haynes, Amrita Mazumdar, Armin Alaghi,
Magdalena Balazinska, Luis Ceze, Alvin Cheung
Paul G. Allen School of Computer Science & Engineering
University of Washington, Seattle, Washington, USA
{bhaynes, amrita, armin, magda, luisceze, akcheung}@cs.washington.edu

http://lightdb.uwdb.io

ABSTRACT

We present the data model, architecture, and evaluation of LightDB, a database management system designed to efficiently manage virtual, augmented, and mixed reality (VAMR) video content. VAMR video differs from its two-dimensional counterpart in that it is spherical with periodic angular dimensions, is nonuniformly and continuously sampled, and applications that consume such videos often have demanding latency and throughput requirements. To address these challenges, LightDB treats VAMR video data as a logically-continuous six-dimensional light field. Furthermore, LightDB supports a rich set of operations over light fields, and automatically transforms declarative queries into executable physical plans. We have implemented a prototype of LightDB and, through experiments with VAMR applications in the literature, we find that LightDB offers up to 4× throughput improvements compared with prior work.

PVLDB Reference Format:
Brandon Haynes, Amrita Mazumdar, Armin Alaghi, Magdalena Balazinska, Luis Ceze, Alvin Cheung. LightDB: A DBMS for Virtual Reality Video. *PVLDB*, 11 (10): 1192-1205, 2018.
DOI: https://doi.org/10.14778/3231751.3231768

1. INTRODUCTION

Over the last several years, advances in computing, network hardware, and display technologies have generated increased interest in immersive 3D virtual reality (VR) video applications. Augmented and mixed reality (AR and MR, respectively) applications, which intermix 3D video with the world around a viewer, have gained similar attention. Collectively, these virtual, augmented, and mixed-reality (VAMR) applications have become mainstream and widely deployed on mobile and other consumer devices.

Managing VAMR data at scale is an increasingly critical challenge. While all video applications tend to be data-intensive and time-sensitive, VAMR video applications are often particularly so. For example, recent VR light field cameras, which sample every visible light ray occurring within some volume of space, can produce up to a *half terabyte per second* of video data [42, 47]. For spherical panoramic VR videos (a.k.a. 360° videos), encoding one stereoscopic frame of video can involve processing up to 18× more bytes than an ordinary 2D video [30].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 44th International Conference on Very Large Data Bases, August 2018, Rio de Janeiro, Brazil.
Proceedings of the VLDB Endowment, Vol. 11, No. 10
Copyright 2018 VLDB Endowment 2150-8097/18/6.
DOI: https://doi.org/10.14778/3231751.3231768

AR and MR 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 specialized *VAMR systems* have been introduced for preparing and serving VAMR video data (e.g., VRView [71], Facebook Surround 360 [20], YouTube VR [75], Google Poly [25], Lytro VR [41], Magic Leap Creator [43]), with the goal of enabling developers to easily implement their applications. Such systems, however, manage VAMR video as if it were ordinary 2D video, which results in conceptual and technical difficulties that we call the *VAMR impedance mismatch*: developers who use current VAMR systems have to consider details about data in its physical 2D format, and must manually account for factors such as spherical projections (e.g., [11, 62]), angular periodicity, video codec idiosyncrasies (e.g., [48]), and nonuniform sampling [9]. This leads to brittle implementations that intermix application logic with the plumbing required to address this impedance mismatch. Not only that, unlike traditional video applications (e.g., surveillance monitoring and object identification) that process the input iteratively by video frames, VAMR applications (e.g., visualization and games) focus on the user, and need to reason about the user's current viewpoint, which further complicates their development using today's view-agnostic video processing systems.

Further compounding this impedance mismatch, current VAMR systems are fixed to a *particular* 2D video layout. For instance, we are aware of no current 360° system that is able to accept light field data (we describe light fields in Section 2), 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 VAMR impedance mismatch, we develop a system that treats all types of VAMR video in a logically unified manner. We introduce 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 VAMR video that have been proposed [11, 36, 62]. Modeling VAMR videos as TLFs allows developers to express their video operations as *declarative queries* over TLFs, and

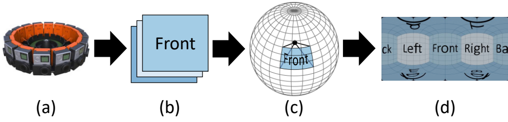


Figure 1: A typical 360° video ingest pipeline. In (a-b), an input camera rig generates overlapping 2D images in every direction. In (c-d) the 2D images are stitched into a spherical representation and then equirectangularly projected onto a 2D plane [30].

decouples the intent of a query from the plumbing and manual optimizations that developers need to implement when using existing VAMR systems and 2D video processing frameworks such as FFmpeg [8] and OpenCV [53]. As we show in this paper, declarative queries also offer the opportunity to introduce query optimization techniques that improve VAMR video workload performance.

To explore these ideas, we have built a prototype system called *LightDB*. LightDB is a new database management system designed to handle the storage, retrieval, and processing of both archived and live VAMR video. It includes an implementation of our novel data model over TLFs, a logical algebra, data storage, query optimization, and execution components, along with a language that allows developers to easily write declarative queries for VAMR workloads.

LightDB builds on recent work in multimedia [6, 31, 39, 58] and multidimensional array processing [7, 12, 55, 54]. It combines the state of the art in array-oriented systems (e.g., multidimensional array representation, tiling) with recently-introduced optimizations by the the multimedia and graphics communities such as motion-constrained tile sets [48], and light field representations [36].

In addition to an implementation of the TLF data model, LightDB exposes a declarative query language, *VRQL*, for developers to use. LightDB automatically selects an execution strategy 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 VAMR systems.

In summary, we make the following contributions:

- We introduce the temporal light field (TLF) data model, which unifies various physical forms of VAMR data under a single logical abstraction (Sections 2 and 3).
- We introduce a logical algebra and query language (*VRQL*) designed to operate over TLFs, and describe real-world workloads using VRQL (Section 3).
- 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 (Section 4).
- 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 $4\times$ increase in FPS for real-world workloads (Section 5).

2. BACKGROUND

To support a broad range of VAMR applications, LightDB ingests and processes two types of VAMR videos: spherical panoramic (360°) videos and light fields. We discuss the details of each in this section.

Spherical Panoramas. One popular form of VR videos are *spherical panoramic videos* (a.k.a 360° videos). These videos allow a viewer, while wearing a head-mounted display or using a mobile device, to observe a scene from a fixed location at any angle. 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 1, 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-dimensional construct, the typical approach projects each sphere onto a two-dimensional plane using a *projection function* and then applies 2D video compression. Common projections are equirectangular (ER) [62], cubic [62], or equiangular cubic [11].

In addition, 360° images and videos may be monoscopic or stereoscopic. Stereoscopic videos 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* [63]) and delivered to a viewer. The viewer uses the depth information to generate and render stereoscopic spherical images locally. We describe below how depth map metadata is embedded in a video.

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 in space. Obviously, this requires knowing the location and orientation of a viewer.

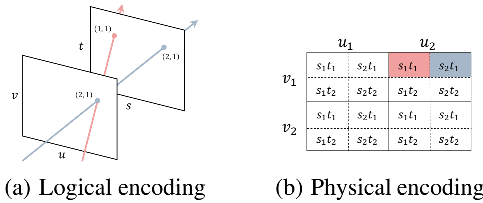
To enable such flexibility, a light field is a function that, for any point in a given volume and for any viewing direction, returns the color and amount of light flowing into a user's eye. Building a light field function requires a matrix of cameras to sample all of the light flowing through a volume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

One method of encoding light field data is to use a *light slab* [36], 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(a), adapted from Levoy & Hanrahan [36], shows the uv and st planes of a single slab.

As in 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 by LightDB and shown in Figure 2(b), encodes each light ray color as an array of arrays [36]. 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(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(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.

Video Encoding & Streaming. Both types of VAMR video described above are ultimately encoded using two-dimensional codecs to reduce the amount of storage required. Modern video codecs accept as input *video frames* that are temporal samples of visual data, generally at *frame rates* of 30-150 frames per second. Each frame is an independent image sampled at a point in time.



(a) Logical encoding (b) Physical encoding

Figure 2: 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.

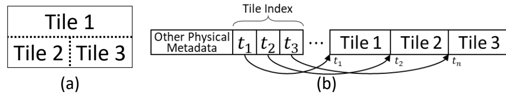


Figure 3: In (a), a frame is subdivided into three independently-decodable tiles. Subfigure (b) shows its corresponding physical representation, where an index points to the offset of each tile.

Video codecs (e.g., H264 [73], HEVC [69]) compress each frame as an intra-frame (a.k.a. *keyframe*) or *predicted frame*. Keyframes are compressed in isolation, and may be decoded independently without reference to any other frame. Predictive frames have dependencies on other frames, which often contain redundant visual information, to improve compression performance. Each predicted frame must be decoded in conjunction with its dependent frames.

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* (sometimes called *slices*) within a frame. While this technique—called *motion constrained tile sets*—reduces compression performance, 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 3, a *tile index* allows for rapid identification of the data region associated with each tile.

Many video codecs 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. GOPs are an important part of *adaptive streaming*, which varies the quality of each GOP delivered to a client based upon its current network conditions [65].

Finally, rather than streaming raw encoded video streams to clients, videos are typically "muxed" into files such as the MPEG-4 Part-14 (MP4) [32] or WebM/Matroska [70] 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 camera perspectives), and allow for data indexing. LightDB uses these features to improve query performance (see Sections 4.1, 4.2 and 4.4).

LightDB extends the widely-supported MP4 file format, both internally and for sending/receiving data to/from external sources. As illustrated in Figure 4, an MP4 file contains 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. For example, the `trak` atom holds metadata about a media stream (e.g., its codec) and a pointer to media data. This media data may be embedded inside a `mdat` atom or stored externally.

A MP4 file may be associated with any number of tracks and video streams. For example, for stereoscopic information, LightDB stores visual information for the left and right eyes in separate

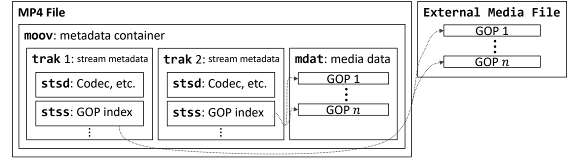


Figure 4: Abridged MP4 layout showing atoms relevant to LightDB. This MP4 file contains a `moov` atom holding media metadata. This includes two `trak` atoms, each containing metadata about a single media stream. A `stss` atom provides a GOP index over media data stored in a `mdat` atom or externally.

video streams. Additionally, if depth map information is available, LightDB embeds it using a separate video stream.

The MP4 format defines one atom (named `stss`) that is useful for efficient query execution in LightDB. This atom contains an index of the GOPs in a video stream, and LightDB uses this to efficiently look up the beginning of any GOP without needing to linearly search through the encoded video data.

To be detailed in Section 4.1, LightDB relies on standard MP4 atoms, an atom drawn from the Spherical Video V2 RFC [67], and a custom atom to store light field-specific metadata.

3. LIGHTDB MODEL

Existing database management systems specialized in the processing of image and video data, including RasDaMan [7], SciDB [12], and Oracle Multimedia [54], model image and video data as multidimensional arrays. These arrays often have three dimensions: x, y, and t. As mentioned in Section 1, we find this model ill-suited for VAMR applications. While standard spatiotemporal dimensions can identify a pixel in a video stream, applications do not reason in those terms. For a VAMR application, the fundamental concepts are *the current location of the viewer* and *the direction in which the viewer is looking*. These concepts, illustrated in Figure 5, are elegantly captured by the light-field abstraction presented in Section 2. Additionally, if a user observes the same object from a different viewing angle, the value of the pixels representing the object must change, which cannot be captured with spatiotemporal dimensions but can be captured with light fields.

3.1 Data Model

LightDB adopts light fields as the fundamental construct in its data model. Because our light fields can change over time, we use *temporal light fields* (TLFs) [2] to represent all data, whether originally ingested as a light field or as a 360° video. Concretely, we model 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 user's position over time while the angular dimensions capture the user's viewing direction.

Definition 3.1 (Nullable temporal light field (TLF))**.** A TLF L_V 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 $V \subseteq \mathbb{R}^4 \times \mathcal{D}_\theta \times \mathcal{D}_\phi$.[1] Application of L at points not in V produces the null token ω.

[1] 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 with $\mathcal{D}_\theta$ and $\mathcal{D}_\phi$. 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.

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., YUV or RGB). 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 6(a), defines a field RB in R that is RED for all $x \leq 0$ and BLUE otherwise:

$$RB_R = \begin{cases} \text{RED} & \text{if } x \leq 0 \\ \text{BLUE} & \text{otherwise} \end{cases} \quad (1)$$

Data objects in LightDB take the form of nullable, temporal light fields. Every TLF, L, in LightDB is 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 6(b) shows RB' as a possible partitioning of RB_R, where one partition contains the RED light rays and another contains the BLUE. Figure 6(c) shows a further subdivision of RB' into six equal-sized volumes with height $\frac{y_1 - y_0}{3}$.

LightDB requires that a TLF's volume and partitions be a hyperrectangle. 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. For a TLF with this flag set, LightDB automatically updates its ending time as new data arrives.

3.2 Algebra

LightDB's query algebra is designed to enable a variety of operations on different types of VAMR 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 LightDB algebra exposes nineteen logical operators for expressing queries over TLFs. We classify them into three broad categories. First, we describe the data manipulation operators used to manipulate TLFs stored in LightDB. Next, we present the ENCODE and DECODE operators, which 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 the LightDB catalog.

As a running example to illustrate the operators, consider the case of live, adaptive 360° video streaming applications that (1) ingests a video from a camera rig, (2) overlays a watermark, (3) performs contrast adjustment, (4) temporally partitions the result into short segments to support adaptive streaming, and (5) encodes each segment into a format suitable for a client device. Figure 7 shows a query plan for this application, and highlights several of the logical operators that we describe in this section.

3.2.1 *Data Manipulation*

The data manipulation operators exposed by LightDB include:

Selection. The SELECT operator derives a "smaller" TLF from its input. For example, the SELECT operator shown in Figure 7 spatially reduces the watermark W to a single point. Additionally, Figure 6(a) illustrates selection over the TLF defined by Equation 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 specifies a predicate over dimensions, with the restriction 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 6(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. For example, the union operator in Figure 7 overlays a watermark on top of an ingested stream. Here the LAST built-in aggregate is used to disambiguate overlapping light rays by preferring the last element among the inputs (i.e., the watermark).

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 7 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:

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

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 [62] 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 LightDB to parallelize the transformation more efficiently.

To enable such parallelization, LightDB allows a developer to optionally specify a neighborhood when executing a MAP transformation. 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

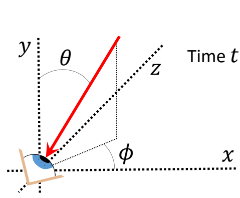
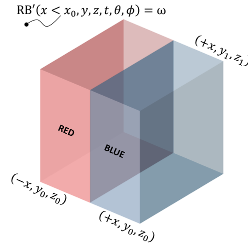
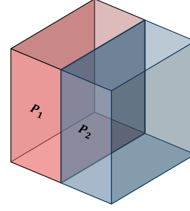


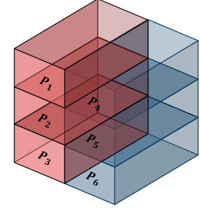
Figure 5: A user positioned at (x, y, z) in three-dimensional space. At time t, rays from every angle reach the user's eye; one ray is shown at (θ, ϕ).



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

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 3.5 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:

$$\text{SUBQUERY}(L, q, m) = \text{UNION}(q(\text{SELECT}(L, V_1)), ..., q(\text{SELECT}(L, V_n)), m)$$

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*

Data flowing into LightDB must always go through a DECODE operator that transforms 360° videos and light field encoded as MP4 files into one of the physical TLF representations to be described in Section 4.1. Similarly, data flowing out of LightDB must go through the reverse ENCODE operator to transform it back into an externally-consumable format.

LightDB also exposes SCAN and STORE operators, which are used to read and overwrite TLFs that are defined in LightDB's internal catalog.

In LightDB, TLFs are immutable and writes are performed by copying video data at the track granularity (see Section 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.

Writes to TLFs are versioned, and version numbers are stored as part of the TLF's metadata (see Section 4.1). 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.

3.2.3 *Data Definition*

LightDB exposes CREATE and DROP operators that 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). Similarly, the DROP operator removes a TLF from the LightDB catalog and deletes its content from disk.

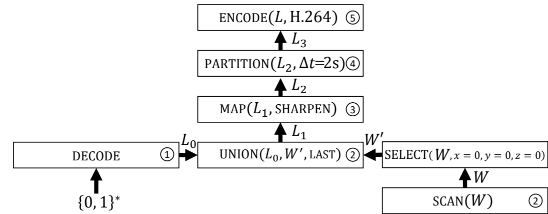


Figure 7: A LightDB plan for a sharpened 360° video with a watermark overlay encoded in two-second partitions.

Table 1: Example expressions using the LightDB algebra

Description	Algebraic Expression
Self-concatenate a 5-second TLF	$\text{UNION}\Big(\text{SCAN}\,(\text{name})\,,\,\text{TRANSLATE}\big(\text{SCAN}\,(\text{name})\,,\,\Delta t = 5\big)\Big)$
Grayscale and H264 encode	$\text{ENCODE}\Big(\text{MAP}\big(\text{SCAN}\,(\text{name})\,,\,\text{GRAYSCALE}\big),\,\text{H264}\Big)$
Sharpen middle third of a TLF	$\text{SUBQUERY}\left(\begin{matrix}\text{PARTITION}\,\big(\text{SCAN}\,(\text{name})\,,\,\Delta\phi = {}^{\pi}/_{3}\big)\,, \\ L \mapsto \begin{cases}\text{MAP}(L\,,\,\text{SHARP}) & \text{if } V_\theta(L) = \left[{}^{\pi}/_{3}, {}^{2\pi}/_{3}\right] \\ L & \text{otherwise}\end{cases}\end{matrix}\right)$

Finally, the 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. To be discussed in Section 4.2, TLF indexes may be multi-level and are represented as dense arrays (for temporal or angular indexes) or spatial R-trees.

3.3 Algebra Expressions

LightDB operators can be composed into expressions in the same way as relational algebra operators. Table 1 shows a few simple examples. We present more complex applications in Section 3.5.

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.

3.4 Query Language

LightDB includes a declarative query language called *VRQL*. VRQL allows developers to describe queries without the need to be concerned with the underlying complexities of the video data, how 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 plan to extend it to other languages.

In VRQL, developers write queries over TLF using functions that correspond to the algebra described previously. For example, a developer would write the following query for the example described at the beginning of Section 3.2 with query plan shown in Figure 7:

```
Union(Decode(filename), Scan("W") >> Select(0, 0, 0))
  >> Map(sharpen) >> Partition(Time, 2)                    (2)
  >> Encode(H264);
```

As is common in functional languages, the streaming operator $g(\alpha)$ >> $f(\beta)$ is used as shorthand for $f(g(\alpha), \beta)$; the two forms are otherwise equivalent and either may be used in a query.

VRQL exposes the operators introduced in Section 3.2 as functions, and convenience functions for common operations (e.g., `transcode` converts a TLF from one codec to another).

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 1:

```
auto tlf = Scan(name);
auto cat = Union(tlf, (tlf >> Translate(Time, 5));
```

Developers may use VRQL to create indices over existing TLFs. 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 may then utilize this index on line 3.

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

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 6(a), a developer would define an anonymous function used in MAP in the following query:

```
auto x = 1;
auto tlf = Create("RB")
  >> Select(Volume{{-x, x}, ...})
  >> Map([](auto &point) { return point.x < 0 ? Red : Blue; });
```

3.5 Applications

In Figure 7, we described a simple query used to ingest, watermark, partition, sharpen, and re-encode an input TLF. We now describe more complex queries drawn from real-world examples in the recent literature.[2] While we are aware of no comprehensive VAMR application benchmark or corpus that could be used to demonstrate the completeness of the LightDB algebra, these applications are representative of the VAMR applications that we have found in this domain.

Predictive 360° Video Tiling. Many streaming video services (e.g., YouTube VR [75]) serve an entire panoramic viewing sphere to client devices. This approach is suboptimal, because at any instant only a small portion of the sphere is displayed in a VR viewer, which generally have a narrow field of view. Recent work [21, 26, 30, 45] has demonstrated substantial savings (up to 75%) in data transfer by degrading the quality of the unimportant areas of the 360° sphere. Developing this query in existing frameworks is tedious and error-prone (see Section 5.1). Using LightDB, a developer can simply express this as the following:

```
Decode("rtp://...")
  >> Partition(Time, 1, Theta, π / 2, Phi, π / 4)
  >> Subquery([](auto& partition) {
       return Encode(partition, is_important(partition)
         ? Quality::High : Quality::Low) })
  >> Store("output");
```

[2]Example output videos produced by LightDB for these applications may be retrieved at `http://lightdb.uwdb.io/examples`.

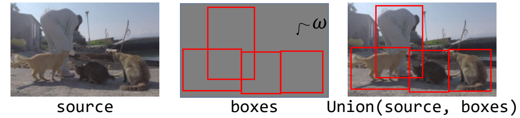


Figure 8: Augmented reality TLFs for one 360° frame

This query subdivides the input TLF into one-second segments and partitions of size $(\frac{\pi}{2}, \frac{\pi}{4})$. Next, the SUBQUERY operator changes the quality of each partition to that given by a function that predicts a user's future orientation. For example, we might use dead reckoning to predict a user's next orientation and encode that partition in high quality (and low quality elsewhere).

Augmented Reality. One augmented reality application that has seen heightened interest in the computer vision [60, 38] and mobile sensing [14, 59] communities involves ingesting a live video stream from a worn camera or mobile device, automatically detecting objects within the field of view, and highlighting them in real-time with a bounding rectangle. In many cases, the object detection function associated with this application is a neural network trained on a particular input video resolution (e.g., 480×480). This workload may be easily expressed in VRQL:

```
auto source = Decode("rtp://...");
auto lowres = source >> Discretize(Theta, 2π/480, Phi, π/480);
auto boxes = lowres >> Map(detect);
Union(source, boxes) >> Store("output");
```

This query lowers the resolution of its input to 480×480, and applies a MAP UDF called $detect$. The UDF applies an object detection algorithm such as YOLO9000 [60] and generates a result that is red at detection boundaries and null otherwise (see Figure 8). Finally, the result is combined with the original input.

Depth Map Generation. Several recent projects have explored real-time depth map generation using parallel or custom hardware [31, 49, 4, 46] 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 360° videos incident to those points [34]. The VRQL query for this process is as follows, where the `DepthMapInterpolation` function implements the logic described in [46]:

```
auto stereo = Union(Scan(L) >> Select(p + i/2),
                    Scan(L) >> Select(p - i/2))
  >> Interpolate(DepthMapInterpolation)
  >> Store("sample");
```

4. LIGHTDB ARCHITECTURE

In this section, we present LightDB's overall architecture together with the details of its core components. LightDB is currently a single-node, single-threaded system. We plan on extending it to support distributed execution in shared-nothing clusters. In our prototype, users submit individual queries as a statement or script that may include variable assignments. LightDB executes each such query as a single transaction with snapshot isolation. We currently disallow queries that overwrite the same TLF more than once.

The major components of LightDB are shown in Figure 9. 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.

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 op-

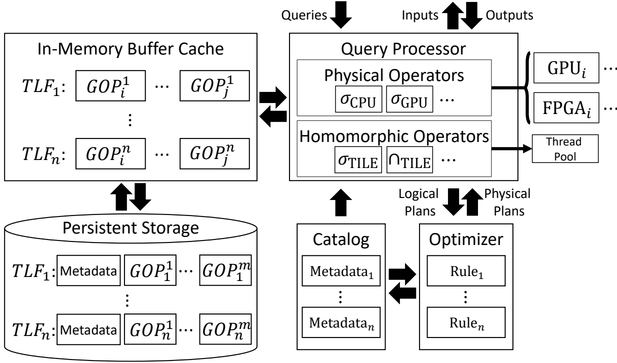


Figure 9: LightDB architecture

timizations, some of which we describe in Section 4.5. LightDB's physical algebra includes operators that directly process encoded video frames, which we call *homomorphic operators*, and operators that process decoded data. LightDB also offers specialized operator implementations for different hardware platforms, which includes GPUs and FPGAs.

In this section, we describe each major LightDB component.

4.1 Physical Organization and Data Storage

In Section 2 we introduced two common encoding methods for VAMR video data: 360° videos (optionally containing depth information) and light slabs. The first efficiently represents visual data incident to a point, while the second is efficient at representing visual data incident to a plane.

LightDB physically represents each TLF using one of these two physical formats: a *physical 360° TLF (360TLF)* or a *physical light slab TLF (SlabTLF)*. Each 360TLF contains one or more 360° videos, with a separate video stream encoded for each non-null spatial point in the TLF. SlabTLFs contain one or more slabs at various positions and orientations, with a separate stream encoded for each.

Both 360TLFs and SlabTLFs 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 360TLF and SlabTLFs 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 360TLFs and SlabTLFs, potentially recursively.

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

- A SlabTLF is converted to a 360TLF 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 360TLF whenever it contains only 360TLFs, and to a SlabTLF when it contains only SlabTLFs.
- When a light slab in a SlabTLF or a 360° video in a 360TLF is no longer visible within its bounding volume, it is dropped.

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

Similarly, a SlabTLF contains a set of light slabs (Section 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. 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.

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. For example, multiple CompositeTLFs can reference the same video file without requiring it be duplicated on disk.

The metadata file is a small (generally less than 20kB) MP4-compliant multimedia container (see Section 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) that contain the properties of the TLF and pointers to associated video streams.

As described in Section 2, LightDB uses standard MP4 atoms to store pointers to separately-stored encoded video streams and the `sv3d` drawn from the Spherical Video V2 RFC [67] to store a 360TLF'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 360TLF, this includes the points at which the TLF is defined. For a SlabTLF, this includes geometry and sampling granularity for each of the slabs. Finally, a CompositeTLF recursively contains two or more `tlfd` atoms as children.

Each `tlfd` atom also contains pointers to video tracks, which store metadata about the underlying video data that support the discrete TLF (see Section 2). For a 360TLF, the `tlfd` atom contains a pointer to one or more video spheres and optional pointers to depth map tracks. SlabTLFs contains one pointer per slab.

To illustrate this structure, we show in Figure 10 an abridged physical layout for the output of the depth map generation query discussed in Section 3.5. This TLF has a single version and associated metadata file `metadata1.mp4`. This file contains a `tlfd` atom that describes a 360TLF defined at two points: $(\boldsymbol{p} \pm \frac{\boldsymbol{i}}{2})$. It is discrete, and so has no view subgraph. It has pointers to two `trak` video tracks, one for each 360° sphere, and also contains a depth map `trak` atom. Each `trak` atom 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, `{stream0,stream1,depthmap}.hevc`.

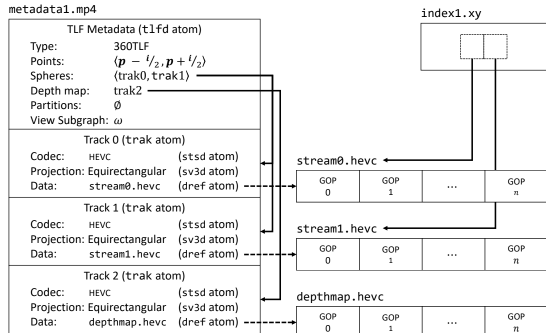


Figure 10: Physical layout of a stereoscopic 360° 360TLF.

4.2 Indexing

In Section 2, we introduced two indices found in encoded VAMR video: a tile index (Figure 3) and a GOP index (Figure 4). When available, LightDB 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 [28] 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 $\text{CREATEINDEX}(L, x, z)$, LightDB utilizes the resulting R-tree for queries of the form $\text{SELECT}(L, x \in [a, b], z \in [c, d], ...)$. As shown in Figure 10, a spatial index is stored as an external file with a name containing its version and covered dimensions (e.g., `index1.xz`).

Next, if a temporal constraint is present in a selection, LightDB considers 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 4) and maps a starting time to the byte offset of the associated GOP. Given a temporal selection (e.g., $\text{SELECT}(t \in [a, b], ...)$), 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 $\text{SELECT}(\phi \in [0, \frac{\pi}{2}])$ might only need to decode Tile 1 shown in Figure 3. Since this index is also used by video decoders, an attempt to drop an angular index results in an error.

4.3 Buffer Pool

During query execution, the query processor (QP) interacts with an in-memory TLF cache (TC) to load data from persistent storage. As shown in Figure 9, the TC contains entries for TLF metadata files, which are parsed from their MP4 representation prior to caching. It also contains a buffer pool for GOPs (see Section 2) that have been recently accessed. Buffering at the GOP granularity improves temporal locality and reduces misses for predictive frame requests. The TC implements a least-recently used eviction policy.

4.4 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 [8] 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 SCAN, ENCODE, DECODE GPU-based operators utilize the hardware-accelerated NVENCODE/DECODE interfaces [50]. The remaining operators each use CUDA [51] kernels to perform their work.

LightDB has a built-in FPGA-accelerated MAP and INTERPOLATE UDF used to generate depth maps as described in Section 3.5.

Finally, LightDB has homomorphic operators (HOps) that perform operations directly on 360TLF or SlabTLF video data *without requiring that it be decoded*. This leads to higher performance — up to 500× faster compared to GPU-based operators (see Section 5). Two categories of HOps exist in LightDB: those that operate over encoded groups of pictures (GOPs; see Section 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 [64]).

To understand the first category, consider a 360TLF or SlabTLF 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., $\text{SELECT}(L, t = [i, j])$, 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. Section 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 3(b), to efficiently identify the relevant bytes without video decoding. For example, consider a 360TLF that has been tiled as shown in Figure 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 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, the operation $\text{UNION}(T_1, T_{23})$ can be performed using this operator.

4.5 Rule-Based Optimization

The LightDB algebra is amenable to several optimizations that improve performance. To demonstrate this, LightDB comprises a rule-based query optimizer that performs two types of optimizations. As future work, we will use these heuristics to implement a cost-based optimizer.

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 query optimizer (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 360° or light field 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 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.

After producing an initial physical query plan, the QO performs further optimizations as follows. First, it makes the following transformations and eliminations:

- 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 $d_1' \geq d_2'$.
- Replace $\text{SELECT}(\text{SCAN}(L), x, y, z)$ with $\text{SCAN}(L)$ if L is a 360° video defined only at (x, y, z). Empty selections are replaced with Ω. Unions with empty inputs (e.g., $\text{UNION}(L, \Omega)$) are replaced with L. If all operands are empty, the union is replaced with Ω.
- 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 11), HOps always outperform operators that execute over decoded video (by up to 500×; see Section 5), even if they require inter-device transfer.

5. EVALUATION

We have implemented a prototype of LightDB in C++ using ~15,000 lines of code. GPU-based operators were implemented using NVENCODE/NVDECODE [50] and CUDA 8.0 [51]. We experimentally evaluate LightDB and compare it to four baseline systems in terms of programmability for complex VR video workloads (Section 5.1) and performance (5.2). In Section 5.3 we show how LightDB is able to utilize hardware accelerators, and in Section 5.4 we detail LightDB operator performance.

Baseline systems. We compare LightDB against OpenCV 3.3.0 [53] and FFmpeg 3.2.4 [8], 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 [50] GPU-based encoding and decoding. OpenCV is built with internal calls to FFmpeg and CUDA 8.0 [51].

We also compare against Scanner [57], 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 [12, 61], 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(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 [17]. 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 [72]. 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).

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 3.5.

For the predictive 360° tiling application, LightDB loads each 360° dataset (Timelapse, Venice, and Coaster) from the file sys-

Table 2: Lines of code required to reproduce the predictive 360° and augmented reality queries described in Section 3.5. 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

tem, 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 [60]. 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 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.

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.

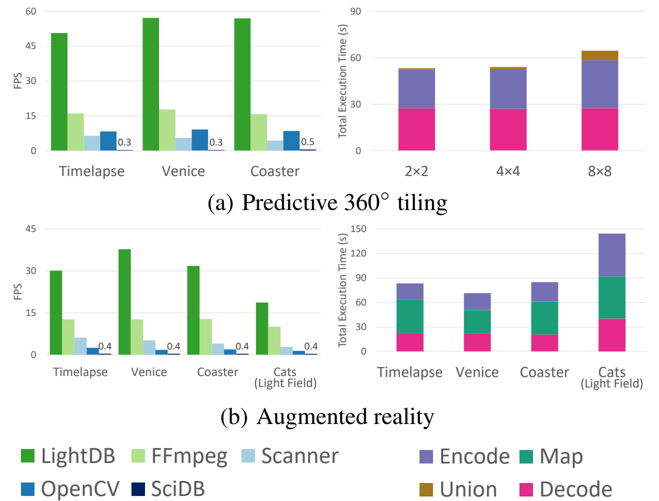


Figure 11: Performance of predictive 360° and AR applications. The y-axes on the left show frames per second (FPS), while the right plot shows LightDB operator contribution to total execution time. Value for SciDB added on top of its bar.

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 3.5 using the Timelapse, Venice, and Coaster datasets.

The throughput for each system and dataset are shown on the left plot of Figure 11(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 4.4) that avoids an expensive additional decode/encode step that is required by the other systems. At runtime and as discussed in Section 4.5, 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 [30, 26, 21, 45], 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 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 11(a) breaks down total query execution time by LightDB operator for the Timelapse dataset and

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: Percent reduction for the predictive 360° query.

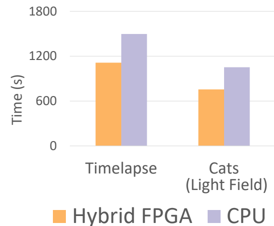


Figure 12: Performance of depth map application

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 3.5 using the 360° datasets as input.

Throughputs for LightDB and comparison systems are shown on the left plot in Figure 11(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 11(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.

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 3.5 using a UDF with a CPU and hybrid FPGA implementation [46] 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 12 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 [46].

5.4 Operator Performance

360TLF Operator Performance. We first examine the individual performance of LightDB operators for 360TLFs using the Timelapse dataset. Figures 13(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) >> $O(\dots)$ >> `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 [62].

Table 4: Video data systems & frameworks. Bolded systems have source available and can execute Section 3.5 applications.

Type	Systems
2D General Purpose	**FFmpeg** [8], **GStreamer** [27], **GPAC** [22] (also supports 360°)
2D Vision	**OpenCV** [53], **OpenIMAG** [29]
2D Analytics	**Scanner** [57], BlazeIt [33], Optasia [40], VideoStorm [76], Rocket [3]
2D Content/Feature Search	VDBMS [5], BilVideo [19], VideoQ [13], AVIS [1], DelaunayMM [18]
2D Metadata Search	VideoAnywhere [66], OVID [52]
2D Presentation	MINOS [15]
Streaming & Transcoding	Morph [23], 360ProbDASH [74]

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 13. 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 13(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 14 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 14(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 14(b), perform at a rate comparable to their 360TLF counterparts (see Figure 13(b)).

Effectiveness of Optimizations. Finally, we evaluate the performance impact of several LightDB optimizations in Figure 15.

The left two plots in Figures 15(a) and 15(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" [16].

The "Self Union" plot illustrates the effect of other query optimizations on LightDB performance. Here, we execute the degenerate query 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 SELECT$(L, [-\infty, +\infty])$.

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 16(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.

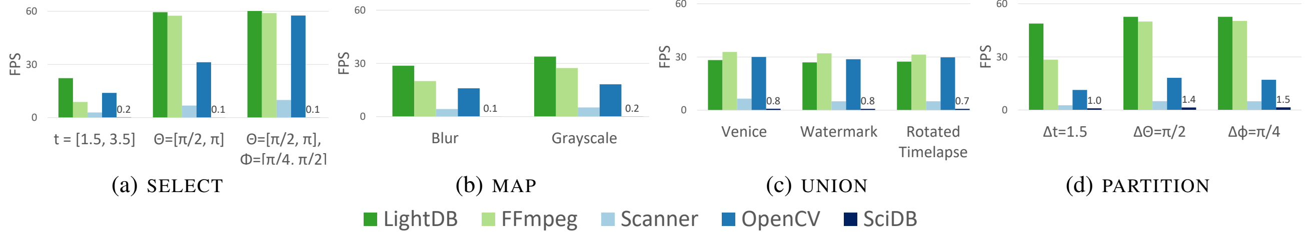


Figure 13: 360TLF operator performance (Timelapse dataset)

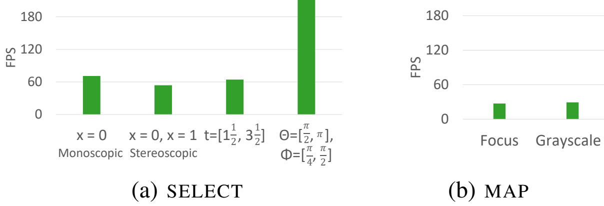


Figure 14: LightDB SlabTLF performance (Cats dataset)

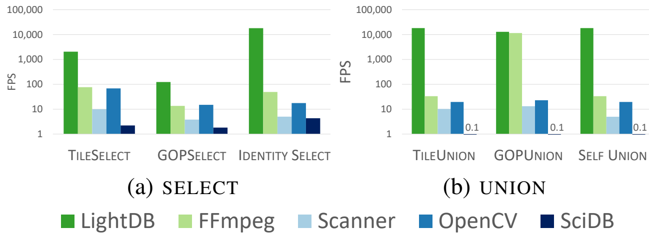


Figure 15: Homomorphic & optimized performance (log scale)

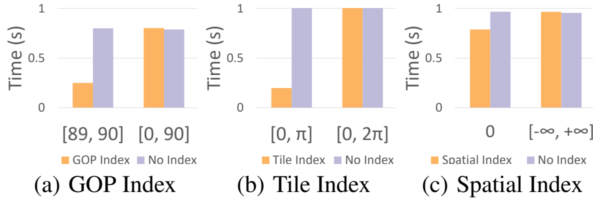


Figure 16: Index Performance

Next, we tiled the Timelapse dataset using the configuration shown in Figure 3. Figure 16(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 16(c), and illustrate that the R-tree yields a modest benefit relative exhaustive search for relevant videos.

6. RELATED WORK

Several systems explore efficient video delivery that targets a single format or workload. Examples include 360° streaming systems [39, 74] and light field image-based rendering systems [6, 58]. LightDB generalizes these and offers a unified declarative query language independent of the underlying physical video format.

Multiple previous systems [37, 40, 56, 3, 76] target 2D video analytics. VideoStorm [76] allow users to express distributed analytical workloads over 2D video (e.g., citywide security feeds), while Optasia [40] also supports declarative queries. Similarly, VAIT [37] offers a small set of predefined queries over large video datasets. Since these systems target 2D video analytics, they force developers to manually map 3D environments onto 2D constructs, which results in rigid applications that are difficult to maintain and evolve.

Other previous video systems, shown in Table 4, variously provide content-based, keyword or metadata, and similarity or feature-based search for 2D video and images. LightDB supports much richer workloads. The GStreamer [27] and Scanner [57] libraries allow for fixed pipelines that are similar to LightDB query plans, but these pipelines are rigid, closely tied to a physical execution strategy, and also require manually mapping 3D constructs onto a 2D equivalent. Finally, the SQL multimedia (SQL/MM) standard [68] extends the SQL specification to a limited set of operations (e.g., cropping, color histograms) over 2D images, but does not support video or user-defined extensions.

The database community has explored the application of array-based data models that are similar to the light field model exposed by LightDB. For example, the RasDaMan [7], SciDB [12], and TileDB [55] 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 virtual and augmented reality video such as continuousness, angular periodicity, and nonuniform projections. Additionally, existing array DBMSs do not support video compression and its idiosyncrasies. As we showed in this paper, the result is dramatically reduced performance.

Other work (e.g., [24, 44, 46]) focuses on capture, stitching, and depth estimation aspects of 360° video. Similar examples exist for light field capture [35]. These efforts are complementary to LightDB, which accepts preprocessed 360° and light field videos from these pipelines and performs further query processing.

In our evaluation, 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 [30, 21, 26, 45]. These applications, however, are dedicated exclusively to the task of 360° video tiling and do not generalize to other workloads as does LightDB. Birklbauer et al. [10] show similar advantages for light field rendering.

7. CONCLUSION

In this paper, we presented LightDB, a DBMS designed to efficiently process virtual, augmented, and mixed reality (VAMR) video. LightDB exposes a data model that treats VAMR 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.

Acknowledgments. This work is supported in part by the NSF through grants CCF-1703051, IIS-1247469, IIS-1546083, IIS-1651489, OAC-1739419, CNS-1563788; DARPA award FA8750-16-2-0032; DOE award DE-SC0016260; a Google Faculty Research Award; an award from the University of Washington Reality Lab; a gift from the Intel Science and Technology Center for Big Data; the Intel-NSF CAPA center; and gifts from Adobe, Amazon, Google, and Huawei.

8. REFERENCES

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

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

[3] G. Ananthanarayanan, P. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha. Real-time video analytics: The killer app for edge computing. *IEEE Computer*, 50(10):58–67, 2017.

[4] R. Anderson, D. Gallup, J. T. Barron, J. Kontkanen, N. Snavely, C. Hernández, S. Agarwal, and S. M. Seitz. Jump: virtual reality video. *TOG*, 35(6):198:1–198:13, 2016.

[5] W. G. Aref, A. C. Catlin, J. Fan, A. K. Elmagarmid, M. A. Hammad, I. F. Ilyas, M. S. Marzouk, and X. Zhu. A video database management system for advancing video database research. In *MIS*, pages 8–17, 2002.

[6] I. Bauermann, Y. Peng, and E. G. Steinbach. RDTC optimized streaming for remote browsing in image-based scene representations. In *3DPVT*, pages 758–765, 2006.

[7] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The multidimensional database system rasdaman. In *SIGMOD*, pages 575–577, 1998.

[8] F. Bellard. FFmpeg. `https://ffmpeg.org`.

[9] N. Birkbeck, C. Brown, and R. Suderman. Quantitative evaluation of omnidirectional video quality. In *QoMEX*, pages 1–3, 2017.

[10] C. Birklbauer, S. Opelt, and O. Bimber. Rendering gigaray light fields. *Computer Graphics Forum*, 32(2):469–478, 2013.

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

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

[13] S. Chang, W. Chen, H. J. Meng, H. Sundaram, and D. Zhong. Videoq: An automated content based video search system using visual cues. In *MMSys*, pages 313–324, 1997.

[14] T. Y. Chen, L. Ravindranath, S. Deng, P. Bahl, and H. Balakrishnan. Glimpse: Continuous, real-time object recognition on mobile devices. In *SenSys*, pages 155–168, 2015.

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

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

[17] X. Corbillon, F. D. Simone, and G. Simon. 360-degree video head movement dataset. In *MMSys*, pages 199–204, 2017.

[18] I. F. Cruz and W. T. Lucas. Delaunay^{mm}: A visual framework for multimedia presentation. In *VL*, pages 216–223, 1997.

[19] M. E. Dönderler, E. Saykol, U. Arslan, Ö. Ulusoy, and U. Güdükbay. Bilvideo: Design and implementation of a video database management system. *MTAP*, 27(1):79–104, 2005.

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

[21] J. L. Feuvre and C. Concolato. Tiled-based adaptive streaming using MPEG-DASH. In *MMSys*, pages 41:1–41:3, 2016.

[22] J. L. Feuvre, C. Concolato, and J. Moissinac. GPAC: open source multimedia framework. In *ICME*, pages 1009–1012, 2007.

[23] G. Gao and Y. Wen. Morph: A fast and scalable cloud transcoding system. In *MM*, pages 1160–1163, 2016.

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

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

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

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

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

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

[30] B. Haynes, A. Minyaylov, M. Balazinska, L. Ceze, and A. Cheung. Visualcloud demonstration: A DBMS for virtual reality. In S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1615–1618. ACM, 2017.

[31] S. Heinzle, P. Greisen, D. Gallup, C. Chen, D. Saner, A. Smolic, A. Burg, W. Matusik, and M. H. Gross. Computational stereo camera system with programmable control loop. *TOG*, 30(4):94:1–94:10, 2011.

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

[33] D. Kang, P. Bailis, and M. Zaharia. Blazeit: An optimizing query engine for video at scale (extended abstract). In *SysML*, 2018.

[34] C. Kim, A. Hornung, S. Heinzle, W. Matusik, and M. H. Gross. Multi-perspective stereoscopy from light fields. *ACM Trans. Graph.*, 30(6):190:1–190:10, 2011.

[35] B. Krolla, M. Diebold, B. Goldlücke, and D. Stricker. Spherical light fields. In *BMVC*, 2014.

[36] M. Levoy and P. Hanrahan. Light field rendering. In *SIGGRAPH*, pages 31–42, 1996.

[37] S. Liu, J. Pu, Q. Luo, H. Qu, L. M. Ni, and R. Krishnan. VAIT: A visual analytics system for metropolitan transportation. *ITS*, 14(4):1586–1596, 2013.

[38] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. E. Reed, C. Fu, and A. C. Berg. SSD: single shot multibox detector. In *ECCV*, pages 21–37, 2016.

[39] X. Liu, Q. Xiao, V. Gopalakrishnan, B. Han, F. Qian, and M. Varvello. 360° innovations for panoramic video streaming. In *HotNets*, pages 50–56, 2017.

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

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

[42] Lytro Immerge. `https://www.lytro.com/press/releases/`

lytro-brings-revolutionary-light-field-technology-to-film-and-tv-production-with-lytro-cinema.

[43] Magic Leap. https://www.magicleap.com.

[44] K. Matzen, M. F. Cohen, B. Evans, J. Kopf, and R. Szeliski. Low-cost 360 stereo photography and video capture. *TOG*, 36(4):148:1–148:12, 2017.

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

[46] A. Mazumdar, A. Alaghi, J. T. Barron, D. Gallup, L. Ceze, M. Oskin, and S. M. Seitz. A hardware-friendly bilateral solver for real-time virtual reality video. In *HPG*, pages 13:1–13:10, 2017.

[47] T. Milliron, C. Szczupak, and O. Green. Hallelujah: The world's first Lytro VR experience. In *SIGGRAPH*, pages 7:1–7:2, 2017.

[48] K. M. Misra, C. A. Segall, M. Horowitz, S. Xu, A. Fuldseth, and M. Zhou. An overview of tiles in HEVC. *Journal of Selected Topics in Signal Processing*, 7(6):969–977, 2013.

[49] R. A. Newcombe, D. Fox, and S. M. Seitz. Dynamicfusion: Reconstruction and tracking of non-rigid scenes in real-time. In *CVPR*, pages 343–352, 2015.

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

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

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

[53] OpenCV. Open Source Computer Vision Library. https://opencv.org.

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

[55] S. Papadopoulos, K. Datta, S. Madden, and T. G. Mattson. The TileDB array data storage manager. *PVLDB*, 10(4):349–360, 2016.

[56] Y. Peng, H. Ye, Y. Lin, Y. Bao, Z. Zhao, H. Qiu, Y. Lu, L. Wang, and Y. Zheng. Large-scale video classification with elastic streaming sequential data processing system. In *LSVC*, 2017.

[57] A. Poms, W. Crichton, P. Hanrahan, and K. Fatahalian. Scanner: Efficient video analysis at scale. In *SIGGRAPH*, 2018 (to appear).

[58] P. Ramanathan, M. Kalman, and B. Girod. Rate-distortion optimized interactive light field streaming. *IEEE Transactions on Multimedia*, 9(4):813–825, 2007.

[59] X. Ran, H. Chen, Z. Liu, and J. Chen. Delivering deep learning to mobile devices via offloading. In *Network@SIGCOMM*, pages 42–47, 2017.

[60] J. Redmon and A. Farhadi. YOLO9000: better, faster, stronger. In *CVPR*, pages 6517–6525, 2017.

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

[62] D. Salomon. *The Computer Graphics Manual*. Texts in Computer Science. Springer, 2011.

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

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

[65] M. Seufert, S. Egger, M. Slanina, T. Zinner, T. Hoßfeld, and P. Tran-Gia. A survey on quality of experience of HTTP adaptive streaming. *IEEE Communications Surveys and Tutorials*, 17(1):469–492, 2015.

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

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

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

[69] G. J. Sullivan, J. Ohm, W. Han, and T. Wiegand. Overview of the high efficiency video coding (HEVC) standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1649–1668, 2012.

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

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

[72] T. Wang, J. Zhu, N. K. Kalantari, A. A. Efros, and R. Ramamoorthi. Light field video capture using a learning-based hybrid imaging system. *TOG*, 36(4):133:1–133:13, 2017.

[73] T. Wiegand, G. J. Sullivan, G. Bjntegaard, and A. Luthra. Overview of the H.264/AVC video coding standard. *TCSVT*, 13(7):560–576, 2003.

[74] L. Xie, Z. Xu, Y. Ban, X. Zhang, and Z. Guo. 360probdash: Improving qoe of 360 video streaming using tile-based HTTP adaptive streaming. In *MMSYS*, pages 315–323, 2017.

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

[76] H. Zhang, G. Ananthanarayanan, P. Bodík, M. Philipose, P. Bahl, and M. J. Freedman. Live video analytics at scale with approximation and delay-tolerance. In *NSDI*, pages 377–392, 2017.