Optimizing Logging and Recovery of Video Data in DBMSs

Joseph Komskis
Isaac Weintraub
Georgia Institute of Technology
Atlanta, Georgia, USA

1 THE PROBLEM

Our project seeks to explore and optimize how modifications to video data can be correctly and efficiently logged and recovered in a database management system. When video data is stored directly in a general purpose relational database, significant overhead is incurred during logging and recovery whenever the data is modified if physical logging is used. When these general purpose databases store videos or other multimedia data, they tend to be stored opaquely as BLOBs. Whenever the BLOB is modified, the database must track in the log file the physical, byte-level changes in the data. This may be acceptable in video-related workloads that are read-mostly and perform few changes to the data once it is in the database, but some applications may need to process the frames of a video by adding filters or changing the resolution before running typical workloads on it. In these cases logging and recovery will be crucial to the overall performance of the DBMS. In our project we seek to explore ways to optimize how changes to video data are logged and recovered by treating video data as a first-class citizen in a storage engine and leveraging logical logging that takes advantage of the storage engine's knowledge of the format of video data and the modifications performed on it.

2 PROBLEM IMPORTANCE

As multimedia data has become more pervasive in recent years, proper and efficient management of video data has become increasingly important. Video data is often used in applications such as machine learning and virtual reality. Databases are often employed to help manage large amounts of data, and while databases can be used to store video data, most tend to be agnostic to the data stored in them. This can lead to challenges and inefficiencies when storing video data due to its large size and the operations performed on the data. For example, if only a small portion of a video is needed, a database that stores the video as a BLOB must read and return the entire video. In contrast, a DBMS that understands how a video is split into frames may be able to return only the relevant frames to the user.

3 RELATED WORK

Several video DBMSs have been developed that are designed for efficient processing of video data. However, few of these video DBMSs focus on the issue of logging and recovery of modifications made to the data stored in these databases. VisualWordDB [7] use a dedicated video store for video data and a relational store for other data. Video data is read-only. If a video needs to be modified for processing, a new view is created from the existing video, which may be stored on disk to serve future requests. Similarly, in BilVideo [6] raw video data are stored separately from video data features. VDBMS [5] does not make as strong of a distinction between the storage of video data and metadata, but does specify that the database buffer pool is split between the database buffer area and the streaming area where requests for streams are serviced. In LightDB [8], every video is read-only, immutable, and versioned. Whenever a video is written, a new copy of the video is made and updated as needed. This has the added benefit of providing snapshot isolation during query evaluation.

VDMS [10] is an open source project from Intel that efficiently supports visual analytics and machine learning on visual data. VDMS stores and retrieves visual data through a custom library, which supports storing data in a new format that is more machine friendly and efficient for data analysis and machine learning workloads. Metadata is stored in a high performance graph database that leverages persistent memory. VStore [11] is another video store built to support video analytics. Unlike other projects, VStore determines video formats and parameters from the needs of the analytics workloads run on the data. It attempts to find the fastest format and parameters for data consumption given the accuracy needs of the operators run on the video data. VStore uses LMDB to store video data, similar to how our project leverages Petastorm for this purpose.

4 IMPLEMENTATION

Our project seeks to solve this problem by designing and implementing a protocol that can optimize the logging and recovery of changes to video data in a DBMS that is aware of video data and its storage format. We have implemented

our project using Python, primarily because it has a strong ecosystem of existing tools we can leverage. Using similar technologies to those used in EVA [2], we have used Petastorm [4] as a storage engine for our video data. Petastorm is a library built on top of Apache Spark [1], a distributed processing system used for big data and analytics workloads. Petastorm stores data in Apache Parquet files, and supports directly supplying that data to machine learning frameworks such as Tensorflow and PyTorch. We use OpenCV [3] for processing videos. With OpenCV we can apply various modifications to videos such as color filters, blurs, resizing, and so on. We found these tools integrate into Python programs quite easily.

We now discuss our progress on the project thus far.

4.1 Limitations & Surprises

When we started implementing a logging and recovery scheme based on physical logging, we ran into several surprises in using Petastorm as a storage engine. First, we learned that Petastorm does not support in-place updates to datasets. The only way to write to a dataset is to append to an existing dataset or overwrite it entirely. Additionally, Petastorm does not provide any guarantees on the atomicity of writes. When writing is done in "overwrite" mode, Petastorm first deletes the existing dataset, then writes the new data to the dataset. If the program crashes during this process, the old data will be lost and whatever new data was written will remain. These limitations seem to come from Spark, rather than from Petastorm itself. Given that Spark and Petastorm are primarily used in big data and analytics environments, where data is typically inserted or appended once and rarely, if ever, modified, these limitations are reasonable for those workloads. This did impact our first logging and recovery protocol, as we discuss in the next section. When developing a more optimized protocol, we were able to optimize for these limitations and improve performance.

Initially, we planned for our logging and recovery protocol to be similar to the ARIES protocol used in a general purpose database. As we implemented the protocol we found it diverged from ARIES in multiple ways. This was mostly due to the fact that, unlike a general purpose database, Petastorm does not have any kind of buffer pool. All reads and writes are done directly from and to disk. Thus, our protocol had to be adapted to account for these differences. ARIES was still useful as a set of guiding principles. When we later implemented a buffer manager on top of Petastorm and implemented a new logging and recovery scheme using it the protocol became more similar to ARIES.

4.2 Naive Logging and Recovery Protocol

To establish a baseline for performance, we developed several components to implement a naive logging and recovery protocol based on physical logging. The transaction manager component acts as a new entry point to making updates to objects and leverages the storage engine to persist them. The transaction manager provides transaction semantics and provides consistency, atomicity, and durability for operations performed in a transaction. For our project, we consider concurrent transactions and isolation between them to be out of scope. The log manager component is also used by the transaction manager. It writes log records to the log file and parses them during recovery time. Lastly, we created an update processor component which, given a raw video frame and a desired operation, will apply the operation to the frame and return it. This component leverages OpenCV to process the frames.

The logging protocol is as follows. When a transaction starts, the log manager writes a log record to the log and the transaction manager creates a new directory in the file system for the transaction. This directory is a private workspace where the transaction can store the before and after versions of video files when updating them. We did this rather than store the data directly in the log file to keep the log file more compact and because it made the protocol simpler to implement and reason about.

When a transaction performs an update, the transaction manager will read the video from Petastorm and save it to the transaction's private directory. This is done to support rolling back the transaction later in case the system crashes before the transaction commits. While saving the video, the update processor takes each frame and produces the updated frames. The updated video is also saved to the transaction's private directory so it can subsequently be read back to save it to Petastorm. Once the before and after image of the video are persisted, the log manager writes a log record that includes the transaction ID and paths to the before and after images. Once the record is written, the updated video is immediately rewritten to Petastorm to persist it. Since Petastorm does not support in-place updates, we are forced to save and rewrite the entire video on each update, regardless of whether the update is modifying every frame or only one frame.

Committing a transaction under this protocol is simple: the log manager writes a record acknowledging the commit, and then the transaction manager deletes the transaction's private directory.

If a transaction must be aborted, all updates must be undone since they are all immediately persisted to Petastorm. The log manager goes backwards through the update records for the transaction and writes each before image back to Petastorm. Once this is complete it writes an abort record to

the log to indicate the abort is completed. This protocol does not write compensation log records to the log for simplicity. Since all the update records are idempotent, if a crash occurs during the abort process we can simply rollback all the updates again at a small performance penalty.

Recovery is simplified since all updates are immediately persisted to Petastorm. Our recovery is made up of analysis and undo phases similar similar to those in ARIES. In the analysis phase the log records are read in order from the beginning and the latest log sequence number from each transaction is tracked. Commit and abort records are only written once a transaction has finished committing or aborting, so if one of these records is found that transaction can be removed from the list of active transactions. In the undo phase, every active transaction is rolled back as in the abort process.

Our logging and recovery protocols currently do not support any checkpointing. Given that we expect writes in a video analytics system to be relatively infrequently compared to a general purposes database, the log grows quite slowly and thus checkpointing is less critical. The majority of a transaction's storage footprint is in its private directory and is deleted upon committing or aborting, so a transaction wastes relatively little space once it is completed. Upon the completion of the recovery algorithm, the log file and all transaction directories could be deleted to fully clean any wasted space from old transactions.

4.3 Buffer Manager

When looking for ways to optimize our naive protocol, we found implementing a buffer manager on top of Petastorm had several benefits. First, it required us to find a way to partition Petastorm datasets into smaller chunks of frames in order to efficiently transfer them between memory and disk. Otherwise, it would only be possible to buffer entire videos, which is not feasible for anything longer than a few minutes. Once the datasets were partitioned into chunks, each chunk could be written back to the storage engine without overwriting the other chunks. This means parts of a video could be updated by only reading and writing the relevant chunks. Since each batch of frames could be updated independently of each other, having a partitioned dataset opened the possibility of parallelizing read and write operations. We made use of this when flushing the buffer manger during our tests and benchmarks to increase performance.

Second, a buffer manager allowed us to buffer updated portions of a videos and serve read and write requests from memory before writing the chunk back to disk. The first time a batch of frames is accessed a disk operation is still incurred, but subsequent requests can read the batch from memory. The buffer manager uses a simple LRU eviction policy when a batch needs to be brought from disk and there are no free slots in the buffer manager. Since writing to Petastorm we found was quite slow, being able to delay writes significantly improved performance.

We implemented these changes by first implementing a new version of our existing storage engine interface from EVA to automatically read and write datasets in terms of partitions. We then created a buffer manager component that used this new implementation.

4.4 Buffered Logging and Recovery Protocols

After implementing a buffer manager on top of the storage engine, we created a new transaction manager and logging manager that were aware of the buffer manager and its properties and read and wrote data through the buffer manager instead of going through the storage engine directly. We then implemented a new logging and recovery protocol on top of these new components with three different approaches that use physical, hybrid, and logical logging and recovery respectively. These three protocols are very similar, only differing in how they log and recover changes made to videos.

The physical logging approach logs all redo and undo information physically to the transaction's private workspace. This is done similarly to our initial protocol by storing before and after images of videos. The primary difference is that the buffered protocol only needs to store the before and after versions of the modified batches of frames instead of the entire video.

The logical approach logs all redo and undo information logically in the transaction log. Suppose a user wants to apply a grayscale filter to all frames of a video or apply a blur to a rectangular region of a subset of the frames of a video prior to using the video. Instead of logging the changes to the video at the byte level, the protocol logs the details of the operation (e.g. a grayscale filter or a blur filter and any parameters) and the frames to which it was applied. To achieve this, we will require all operations users wish to perform are predefined, so they can be referred to by name in the log. It is easy to add new operations, and new operations can leverage the functionality of OpenCV or other libraries.

The hybrid approach is a combination of the physical and logical approaches that logs redo information logically and undo information physically. Thus, each transaction stores the before version of each modified batch of frames to its private directory. It also stores in the transaction log the details of update operations and paths to the before versions of modified batches. We implemented this approach because in reality one cannot use logical logging for all operations, as not all operations can be logically undone. For example, suppose a user wants to make a video grayscale or decrease

its resolution. Once a video is made grayscale or downscaled, information about the video is lost, so it is nearly impossible to add color back to it or upscale the video to its original version. Thus, the logical approach described above must fallback to hybrid logging when its update operation is not reversible.

Other than the differences in how updates are undone and redone, the rest of the logging and recovery protocol is the same for all three approaches. When a transaction performs an update, the transaction manager will read the relevant batches through the buffer manager. The physical and hybrid logging approaches will save the original versions of the batches to their private directory to support physical undo. Then, the update is applied to the batches of frames in the buffer manager. The physical approach will also save the updated batches to its directory to support physical redo. Then, the log manager writes to the log file a log record acknowledging the update, which is immediately flushed to disk. Unlike our initial protocol the changed batches are not immediately persisted to Petastorm.

When a transaction commits, the log manager writes a log record acknowledging the commit. Since the changes made to the video are not immediately flushed to disk, if the system crashes before they are flushed then the changes will need to be replayed.

When a transaction aborts, the log manager writes a log record acknowledging the abort. Then, the log manager rolls back the transaction by reading the log records written by the transaction in reverse order and undoing its changes. Since logically undoing changes to a video may not be idempotent, compensation log records are used in all three approaches to track which parts of a transaction have already been rolled back in case a crash occurs during recovery. This can also improve recovery time in the event of a crash during recovery. As in normal operation, batches of frames are read and written through the buffer manager rather than directly to the storage engine. Once all update records for a transaction have been reverted the log manager writes a transaction end record to the log to indicate the transaction has finished executing.

The recovery process is made up of analysis, redo, and undo phases similar to that of ARIES. In the analysis phase all log records are read in order and the latest log sequence number for each transaction is tracked. Commit and transaction end records are written only when a transaction has finished committing or aborting, respectively, so when one of these records is found the transaction can be removed from the list of active transactions. In the redo phase all log records are again read in order, but this time all update and CLR records are replayed to bring the database to the exact state it was in before the crash. For logical and hybrid logging this means reapplying the update operation and for physical logging this means reading the saved after images and rewriting them to the buffer manager. Each frame tracks the LSN of the latest update made to it, which facilitates the redo process so hybrid and logical logging do not incorrectly reapply an update that has already been persisted. In the undo phase, every active transaction found during the analysis phase is rolled back using the process mentioned earlier.

5 SOLUTION VALIDATION

To validate our solution we took many of the storage related components from EVA and implemented our logging and recovery protocol on top of them. This allowed us to test our protocol in isolation.

We wrote unit tests for the various components we implemented and for our protocols to ensure our code is working correctly. For example, we wrote tests to ensure the buffer manager properly reads and writes batches, flushes and discards slots, and evicts slots appropriately when needed. We also wrote tests to ensure that our buffered protocol correctly rolls back transactions upon abort, correctly redoes committed transactions after a crash (when no, some, or all buffer manager slots were persisted before the crash), and correctly handles CLR records if a crash occurs during recovery. From this we are reasonably confident our code works as expected, especially for the paths used in our benchmarks.

Towards this end, we have implemented a feature we are calling "pressure points". Pressure points are conditional edge cases we can enable and disable at runtime while our tests are running. For example, we use pressure points to cause an early return during the recovery process after a CLR is written. This is used to simulate a crash during the recovery process, so we can ensure that during the next recovery phase the CLR is used correctly. These have helped us ensure edge cases in our protocol are correct.

We also validated our implementation worked as we expected by running benchmarks on various aspects of our protocols, as discussed in the next section.

6 SOLUTION EVALUATION

We evaluated the performance of buffered logging and recovery scheme approaches by comparing them against the previous, more naive protocol implemented. From this we measured noticeable performance improvements.

We first benchmarked the performance of logging across three different variables: percent of the video modified, number of updates applied to the video, and length of the video modified. In the percent updated benchmark (Fig. 1), we wanted to see how modifying different amounts of a video would impact performance. In this benchmark we performed one update on a 2.5 minute video, which modified a certain

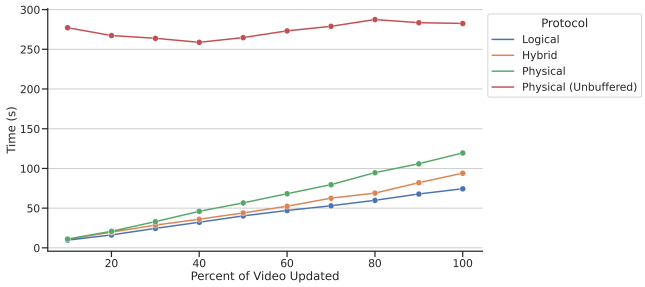

Figure 1: Comparison of the time to apply a single update to a 2.5 minute video modifying varying percentages of the frames of the video.

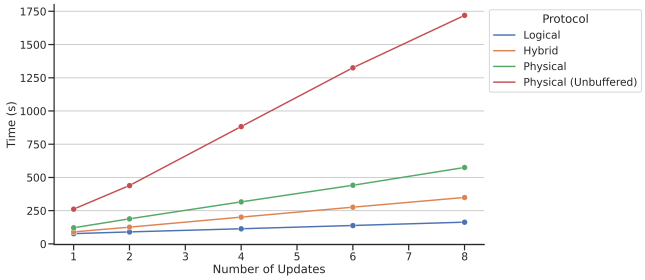

Figure 2: Comparison of the time to apply a number of updates to a updates to a 2.5 minute video.

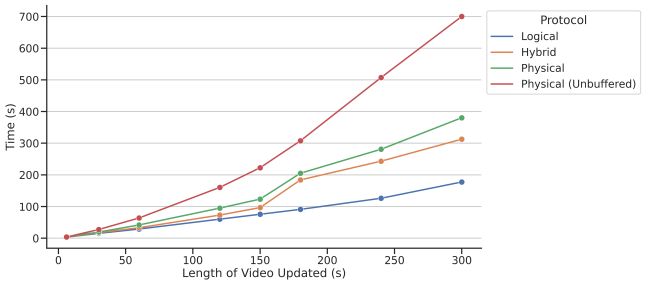

Figure 3: Comparison of the time to apply a single update to videos of varying lengths.

percentage of the video's frames. As expected, in our original protocol the time to apply an update stays roughly the same regardless of how much of the video is modified, since it always saves and overwrites the entire video. We are not entirely sure why the time taken to log an update decreases as the percent modified increases from 10% to 40%, but it may be due fluctuations during benchmarking or in Petastorm. Also as expected, in the buffered protocols the time taken to apply an update increases linearly since more batches of frames must be modified. The increase is more significant for the hybrid and logical approaches since these protocols must apply the update and save versions of batches of frames to the disk, incurring further penalty. This overhead is negligible when small portions of the video are updated but becomes significant as larger portions of the video are updated.

In the number of updates benchmark (Fig. 2), we wanted to measure the performance impact of applying multiple updates consecutively to a video. In this benchmark we update every frame of a 2.5 minute video, as we believe updating every frame of a video is the most typical update scenario. As the number of updates applied increases, the time taken to process the transaction increases linearly. The increase for logical logging is the least significant of the protocols since it need only apply the updates and log the update operation. The increase in time for hybrid logging is more significant since it must save to disk the before version of every batch of frames. Buffered physical logging incurs more overhead still since it must log both the before and after versions of every batch of frames. Our original protocol incurs significantly more overhead than the buffered protocols since for every update it must also save the before and after versions of the video to the disk. Unlike the buffered protocols, however, the original protocol must read and write the video from and to disk for every update, while the buffered protocols can work with the video in memory. Finally, the buffered protocols are able to flush the modified batches concurrently to disk, decreasing the time needed to write the video back to disk.

In the video length benchmark (Fig. 3), we wanted to see how well our protocols scaled with larger videos. In this benchmark, we applied one update to every frame of videos with lengths from 6 seconds to 5 minutes. As expected, the overhead for our naive protocol is much higher than the buffered protocol since it cannot buffer parts of the video in memory. The buffered protocol's time to update starts to increase more significantly past 150 seconds of video. This is because the buffer manager was configured to hold about 150 seconds of video, so increases beyond that means batches of frames start getting swapped in and out to disk, incurring additional overhead. For shorter videos the difference in overhead is negligible, but is much more pronounced for longer videos. One surprise from this benchmark is that the time to apply the update in the naive protocol almost seems to increase quadratically with the increase in video length. We believe this may be because of a constant amount of overhead in the protocol that makes the graph appear quadratic. If time allowed for testing longer videos, we expect the behavior of the graph would be more linear. Another surprise is the sudden jump in time for logical and hybrid logging from the 150 second video to the 180 second video. We believe this is due to the additional overhead from the buffer manager starting to evict frames when it is full.

Next, we benchmarked the time taken to perform recovery of a single committed or aborted transaction with a varying

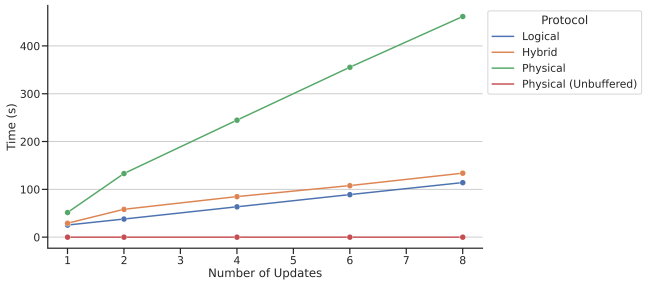

Figure 4: Comparison of the time to redo a transaction that applied a varying number of updates to a a 2.5 minute video.

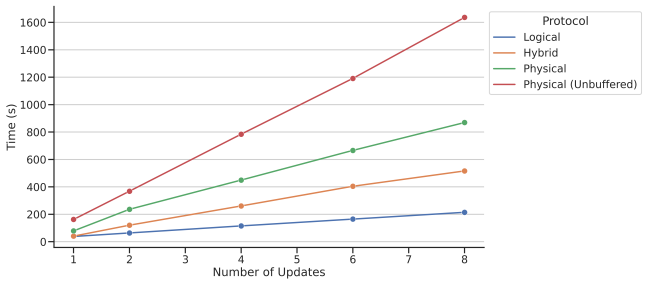

Figure 5: Comparison of the time to redo and undo a transaction that applied a varying number of updates to a 2.5 minute video.

number of updates. In these benchmarks we created a single transaction where each update modified every frame of a 2.5 minute video. In the first case (Fig. 4) we committed the transaction, simulated a crash, then measured the time taken to redo the transaction. As expected, since the naive protocol immediately persists its changes and has no redo phase, recovering committed transactions is almost instantaneous. Buffered hybrid and logical logging have almost the same performance since they both utilize logical redo. Hybrid logging has slightly higher overhead than logical logging, likely due to the larger and more complex log records it parses during the recovery process. Buffered physical logging is significantly slower than the hybrid and logical approaches, since it must redo changes physically, which involves reading the new version of every modified batch of frames from disk.

In the second case (Fig. 5) we simulated a crash before commit, then measured the time taken to redo and undo the transaction. Buffered logical logging has the lowest overhead since it redoes and then undoes the transactions effects logically without having to read large amounts of data from disk. Hybrid logging incurs additional overhead since it must undo changes physically by reading the original frame batches from disk and rewrite them to the buffer manager. Buffered physical logging incurs still more overhead since it must both redo and undo all changes physically, incurring much more disk I/O. Finally, our naive protocol incurs the greatest overhead since during the undo process it immediately persists every undo operation to the storage engine instead of being able to buffer the changes in memory.

The benchmarks were run on an Azure D8as_v4 virtual machine with 8 vcpus, 32 GB of memory, and a 2 TB data drive that provided 7,500 IOPS and 250 MBPS of throughput. This was done for several reasons. First, this virtual machine is a standard size that others could create to mimic our testing environment more closely. Running the benchmarks on our personal laptops would lead to less repeatable experiments. Second, this machine was much more powerful than our laptops, which allowed the benchmarks to be run more quickly. Lastly, running the benchmarks on a cloud machine avoided wear on our own computers.

7 RESOURCES NEEDED

In our unit tests and benchmarking we used videos provided by Visual Road [9], a video data management benchmark. This benchmark uses a game engine to simulate a pseudo-randomly generated city with cars and pedestrians and records the simulation using several cameras placed around the city. The project pre-generated several datasets in multiple resolutions and durations and made them available for download. Since the videos in this dataset tended to be quite large (1 hour), we generated various prefixes of the videos for our purposes.

8 GOALS

After presenting our project update and talking with the professor and TA about our project, we decided to change our goals slightly from our initial goals:

For our 75% goal (unchanged from the progress update) we planned to create a simple storage engine and interface to simulate queries on the database. This allowed us to begin evaluating protocols and optimizations without incurring the software engineering overhead of creating a production-grade DBMS. Using this storage engine, we evaluated a naive logging and recovery as a baseline and identified ways to optimize logging and recovery with video data. This goal is complete.

For our 100% goal we implemented a buffer manager on top of the storage engine and created a new version of the transaction manager and log manager that could make use of the buffer manager. We then implemented a more optimized logging and recovery protocol based on physical logging that used the new buffer manager. This goal is also complete.

For our 125% goal we further optimized our logging and recovery protocol and added support for logical and hybrid logging in our protocol. We also developed and executed benchmarks comparing our different protocols and approaches to quantify the improvements we made. This goal is also complete.

9 FUTURE WORK

There are several ways we could expand or improve our project in the future given more time. First, given additional time, we would have liked to integrate our project into an existing project like EVA to assess its impact in a real-world system. This was originally in our project goals, but after the progress update we decided exploring buffer management would be a more interesting and insightful task. This would give us the opportunity to validate that our logging and recovery scheme is usable and effective in a more fully featured project. It would suggest our solution has legitimate use in a real-word scenario, and not just in an academic setting.

Given more time, it would be interesting to explore other protocols for updating videos. For example, another technique to updating videos would have been to never materialize the updated video and store it in the database. Instead, the DBMS could simply record the operations applied to a video, and reapply them to the original source frames each time the video is read. This would significantly increase write performance, while decreasing read performance depending on how compute intensive the operations are.

It would also be interesting to explore the use of a different storage engine instead of Petastorm. We used Petastorm because that was the project EVA chose to use for storage, but we likely did make use of many of the features it had to offer. There may be other video storage engines that would have been much more efficient for a project such as ours.

Our project read and wrote videos in terms of raw frames. While this worked well enough for our use cases, working with raw frames incurred significant memory and disk usage and bandwidth. With more time, it would have been interesting to explore different ways of encoding video data to try to reduce this footprint and possibly increase performance.

If we had more time for benchmarking, we would have liked to see how changing the size of the buffer manager impacts performance. This was tested to some extent in our video length updated benchmark since the longer videos did not fit entirely in the buffer, but it would have been nice to have a benchmark dedicated to it. This would not be too difficult to add to our project, but running a reliable benchmark for this would be time consuming. Likewise, it would have been nice to extend our benchmarking with mixed workloads to better approximate real usage in contrast to our existing cases which test one operation at a time. We also developed a framework for running benchmark cases under profiling but did not have much time to derive useful insights from it. In the future, having this profiling infrastructure would help empirically identify parts of our implementation to optimize.

REFERENCES

[1] [n.d.]. *Apache Spark - Unified Analytics Engine for Big Data.* Retrieved March 6, 2021 from https://spark.apache.org/

[2] [n.d.]. *EVA: Exploratory Video Analytics System.* Retrieved March 6, 2021 from https://github.com/georgia-tech-db/eva

[3] [n.d.]. *OpenCV.* Retrieved March 6, 2021 from https://opencv.org/

[4] [n.d.]. *Petastorm.* Retrieved March 6, 2021 from https://github.com/uber/petastorm

[5] Walid G. Aref, Ann Christine Catlin, Jianping Fan, Ahmed K. Elmagarmid, Moustafa A. Hammad, Ihab F. Ilyas, Mirette S. Marzouk, and Xingquan Zhu. 2002. A Video Database Management System for Advancing Video Database Research. In *MIS 2002, International Workshop on Multimedia Information Systems, October 10 - November 1, 2002, Tempe, Arizona, USA, Proceedings*. Arizona State University, 8–17.

[6] Mehmet Emin Dönderler, Ediz Şaykol, Umut Arslan, Özgür Ulusoy, and Uğur Güdükbay. 2005. BilVideo: Design and Implementation of a Video Database Management System. *Multimedia Tools and Applications* 27, 1 (01 Sep 2005), 79–104. https://doi.org/10.1007/s11042-005-2715-7

[7] Brandon Haynes, Maureen Daum, Amrita Mazumdar, Magdalena Balazinska, Alvin Cheung, and Luis Ceze. 2020. VisualWorldDB: A DBMS for the Visual World. In *CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p12-haynes-cidr20.pdf

[8] Brandon Haynes, Amrita Mazumdar, Magdalena Balazinska, Luis Ceze, and Alvin Cheung. 2018. LightDB: A DBMS for Virtual Reality Video. *Proceedings of the VLDB Endowment* 11, 10 (01 Jul 2018). https://doi.org/10.14778/3231751.3231768

[9] Brandon Haynes, Amrita Mazumdar, Magdalena Balazinska, Luis Ceze, and Alvin Cheung. 2019. Visual Road: A Video Data Management Benchmark. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 972–987. https://doi.org/10.1145/3299869.3324955

[10] Luis Remis, Vishakha Gupta-Cledat, Christina R. Strong, and Ragaad AlTarawneh. 2018. VDMS: An Efficient Big-Visual-Data Access for Machine Learning Workloads. *CoRR* abs/1810.11832 (2018). arXiv:1810.11832 http://arxiv.org/abs/1810.11832

[11] Tiantu Xu, Luis Materon Botelho, and Felix Xiaozhu Lin. 2018. Reinventing Data Stores for Video Analytics. *CoRR* abs/1810.01794 (2018). arXiv:1810.01794 http://arxiv.org/abs/1810.01794