Building a Scalable Distributed Online Media Processing Environment

Shadi A. Noghabi
advised by: Roy H. Campbell, Indranil Gupta
University of Illinois at Urbana-Champaign,
{abdolla2, rhc, indy}@illinois.edu

ABSTRACT

Media has become dominant in all aspects of human lives, from critical applications such as medical, military, and security (e.g. surveillance cameras) to entertainment applications such as social media and media sharing websites. Billions of massive media objects (e.g., videos, photos, documents, etc.) are generated every second with high diversity among them (in terms of sizes and formats). These objects have to be stored and retrieved reliably, with low latency and in a scalable while efficient fashion. Additionally, various types of processing are done on media objects, from simple compressions and format conversion, to more complex machine learning algorithms detecting certain patterns and objects.

Existing large-scale storage and processing systems face several challenges when handling media objects. My research focuses on building an unified storage and processing environment tailored specifically for media objects, while maintaining high efficiency and scalability. I have built a scalable, load-balanced, efficient storage system optimized for media objects based on their unique access patterns. Currently, I am working on developing an efficient media processing system and integrating these two systems into one framework.

Keywords

Media Processing, Distributed Storage, Online Processing

1. INTRODUCTION

During the past decade, media has become extremely popular. Based on [7], hundreds of hours of videos ($\simeq$ hundreds of GBs) are uploaded per minute in YouTube, the largest Internet video database. These videos have to be processed, reformatted, compressed, verified, and categorized, while being uploaded. Moreover, hundreds of thousands of hours of videos are viewed per minute, from all around the globe. All these videos should be stored and retrieved reliably (with no data loss or unavailability) from a distributed storage at YouTube. This, is only one example of the need for processing and storing large media objects. Many other examples can be found in a wide range of applications from medical imagery and surveillance cameras to social networks and online shopping.

Handling this massive amount of media poses a number of unique challenges. First, a diverse range of media (photos, videos, documents, etc.) with various sizes (from a few KBs to a few GBs), should be processed efficiently at the same time. Second, there is an ever-growing amount of data that needs to be stored, served, and processed in a linearly scalable fashion. Third, in many applications (e.g., machine learning applications), processing is both data and CPU intensive, causing several difficulties in resource scheduling. My research focuses on building an unified online environment tailored specifically for storing and processing these diverse massive media objects, while maintaining efficiency and scalability.

First, we need to store this ever-growing enormous amount of media efficiently. Existing distributed storage systems, including file systems [21, 26] and key value stores [12, 16], face several challenges when serving media objects. These systems impose additional unnecessary overhead (such as rich metadata), and are not efficient in handling both massive GB objects and many small objects, at the same time. Therefore, we need a scalable storage system tailored specifically for diverse media objects.

Additionally, media objects have to be processed in a timely manner. Various types of processing is conducted on media objects including: pattern matching (e.g., detecting pornography), categorization (e.g., tagging photos), correlation detection (e.g., recognizing a burglary in a network of surveillance cameras), compression and resizing, format conversion, and matching media objects (e.g., deduplication and checking copy-write). There has been extensive research on optimizing each of these applications, such as OpenCV and ShapeLogic [3, 4], focusing on a single machine. Recently, the trend has moved toward distributed environments. HIPI, MIPr, and 4Quant [22, 23, 2, 6], have been developed as distributed offline media processing systems. However, in many use cases, especially in sensitive areas such as security and medical, we need real-time processing. Even in less critical areas, such as social networks, delays (more than a few seconds or minutes) could cost millions of dollars. There has been some effort performing real-time media processing by building libraries on top of existing distributed online frameworks [5]. However, current

Proceedings of the VLDB 2016 PhD Workshop, September 9, 2016. New Delhi, India.

online frameworks are not optimized for massive media objects since they do not focus on minimizing data movement. Therefore, we need an online processing framework designed and optimized with the goal of processing large media objects.

2. MEDIA STORAGE

As the first step in my research, I have been focusing on designing an efficient storage for media objects. Handling media poses a number of unique challenges. First, due to diversity in media types, media object sizes vary significantly from tens of KBs (e.g., profile pictures) to a few GBs (e.g., videos). The system needs to store both massive media objects and a large number of small media objects efficiently. Second, there is an ever-growing number of media that need to be stored and served. This rapid growth in requests magnifies the necessity for a linearly scalable system (with low overhead). Third, the variability in workload and cluster expansions can create unbalanced load, degrading the latency and throughput of the system. This creates a need for load-balancing. Finally, data has to be stored and retrieved in a fast, durable, and highly available fashion. For example, when a user uploads a media object in social network, all his/her friends from all around the globe should be able to see the media object with very low latency, even if parts of the internal infrastructure fail. To provide these properties, data has to be reliably replicated in multiple datacenters, while maintaining low latency for each request.

Several systems have been designed for handling a large amount of data, but none of them satisfactorily meet the requirements and scale media processing needs. There has been extensive research into distributed file systems [20, 17, 14, 21, 26]. As pointed out by [11, 15], the unnecessary additional capabilities these systems provide, such as the hierarchical directory structure and rich metadata, are an overkill for a media storage.

Many key value stores [13, 16, 10, 12] have also been designed for storing a large number of objects. Although these systems can handle many small objects, they are not optimized for storing large objects (tens of MBs to GBs). Further, they impose extra overhead for providing consistency guarantees while these are typically not needed for immutable data. Some examples of these overheads include using vector clocks, conflict resolution mechanism, logging, and central coordinators.

A few systems have been designed specifically for large immutable objects including Facebook's Haystack [11] along with f4 [18] and Twitter's Blob Store [25]. However, these systems either become imbalanced (under-utilizing some of the nodes) or do not scale beyond a point.

Thus, through a collaboration with LinkedIn, we developed a scalable load-balanced distributed storage system built specifically for media objects (described below) [1]. Ambry has been running in LinkedIn's production environment for the past 2 years, serving up to 10K requests per second across more than 400 million users. We have published our work **"Ambry: LinkedIn's Scalable Geo-Distributed Object Store"** in SIGMOD 2016 [8].

[1]The project is open-source and the code can be found at [1].

Social networks are one of the biggest sources of media objects, with hundreds of millions of users continually uploading and viewing photos, videos, etc. Typically, these media objects are written once, frequently accessed, never modified, and rarely deleted. We leveraged this immutable read-heavy access pattern of media objects towards Ambry. Ambry is a scalable distributed storage designed for efficiently handling both massive media objects (GBs) and large number of small media objects (KBs). Ambry utilizes techniques such as decentralized design, asynchronous replication, rebalancing mechanisms, zero-cost failure detection, and OS caching. Using these techniques, Ambry provides high throughput (utilizing up to 88% of the network) and low latency (less than 50 ms latency for 1 MB object), while maintaining load balancing amongst nodes.

3. PROCESSING MEDIA OBJECTS

As the next step, I am currently working on building a distributed online processing system, optimized for media objects. Distributed stream processing systems have been designed for processing enormous amount of data in a near real-time fashion [24, 9, 19, 27]. Conceptually, these systems are a great fit for media processing. Stream processing systems are capable of processing massive amount of data in parallel, as the data is generated in a near real-time fashion.

However, existing systems are not optimized for media and incur a lot of data movement. Many of these system include multiple stages of data copy and/or fetching data over the network. Although data movement may not be a dominant factor for processing small data, this is not true for massive media objects.

One of the main causes of data movement is reading and writing data from an external storage system (remote state), as opposed to supporting fault-tolerant locally stored data (local state). For example, for providing exactly-once guarantees[2], Millwheel [9] queries Bigtable [12] on each message it receives to confirm that message has not been processed before. Although an external storage provides faster bootstrap and recovery time, it increases latency, consumes network bandwidth, and can cause denial of service (DOS) for the external storage.

Currently, I am collaborating with LinkedIn on developing Samza, a scalable distributed stream processing system supporting local state. Samza provides fault-tolerant local state by using local database instances in each node, along with a compacted changelog for failure recovery. Each local database instance stores data on disk, providing TBs of space per machine. Additionally, by batching writes and caching popular data in memory, it reaches performance close to an in memory storage. By using local state, we can implement exactly-once guarantees via storing processed message ids locally (with very low latency), and replaying the changelog if failures happen.

We ran an performance benchmark to evaluate the effect of using local state, compared to using remote state. We used two workloads

- ReadWrite: similar to a word count application reading the current count of a specific word and updating.

[2]Exactly-once guarantees means processing each message exactly once, even in presence of failures and late arrivals.

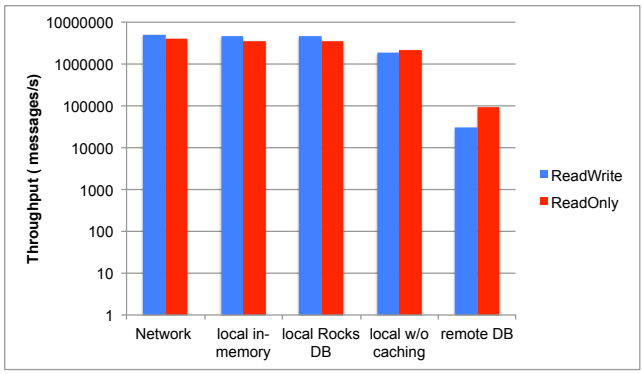

Figure 1: Comparision of different stores in samza under two differnt workloads. This graph compares maximum throughput acivable by the network, local inmemory storage, local rocks db storage, local rocks db with disabling caching, and using a remote storage.

- ReadOnly: similar to a large join of a infinite stream and a table. This workload reads data from the storage and concatenating it with the input message.

We ran the workload on a 4 node cluster of beefy machines, using multiple stores. Based on our initial results, using local state improves throughput up to 100x compared to using remote state, as shown in Figure 1. Using a local Rocks DB with caching enabled reaches almost the performance of a local in memory store and both cases saturate the network. Even with no caching, Rocks DB reaches almost half the throughput of a in-memory store. However, the remote storage is orders of magnitude (up to 100x) slower.

We also measured the latency for each test and it followed a similar pattern. When using a local (Rocks DB or inmemory) store the latency per input data was a few microseconds in all test (8 to 30 microseconds). However, using a remote store this latency was 3 orders of magnitude higher (4-10 milliseconds). This is mainly because of the added delay for going over the network and data copies, and the overhead of providing consistency in a fault-tolerant distributed store (with multiple replicas).

Moreover, we have developed a host affinity mechanism that tries to reduce the failure recovery time by leveraging the state already stored on disk. This mechanism favors brining up failed containers on machines they were placed before, reusing the state stored on disk. Using host affinity we are able to drop the recovery time 5-10x, reducing the recovery time from several minutes to only a few seconds. Also, using host affinity, the overhead of recovery is almost a constant value irrespective of the size of the state to be rebuilt. This is because only the fraction of the state not flushed to disk has to be rebuilt. Therefore, using this mechanism and local state we are able to provide fast failure recovery, close to a remote state, while not incurring the overhead of querying a remote database on each input message processed.

4. FUTURE RESEARCH

Although local state significantly improves performance (specially for applications sensitive to data movement), it is not sufficient for providing an efficient media-processing environment. Due to the large media sizes, the local state may not be enough for storing all the data associated with an application.

For example, assume we have a continuously changing reference set of fraud media, and a fraud detection application that compares recently posted media against a subset of the reference set (based on a similarity metric such as RGB ratio). Using local state, we can partition the reference set based on the similarity metric set across multiple machines; access the set locally; and update the reference set whenever needed. However, if the reference set grows too large (e.g., videos), the local state capacity will not be enough unless by scaling to a bigger cluster with many underutilized machines. I plan to overcome this issue by integrating the distributed storage system and the processing system into one framework. This framework will utilize local state as a cache for storing/retrieving media by offloading data to the storage system, whenever needed.

Additionally, processing even a single massive media object (e.g., a high quality video) in a timely manner can go beyond the capabilities of a single machine. Chunking large media objects into smaller ones and processing chunks in parallel, mitigates this issue. However, chunking introduces many challenges including: rebuilding the large object in a system where data chunks can be processed out of order or infinitely delayed; processing data without losing accuracy; and handling data dependency amongst data chunks. As my future direction, I plan to provide built-in chunking and rebuilding mechanism for large media types, without affecting accuracy.

In a nutshell, as the future direction, I plan to further optimize Samza to efficiently handle diverse media types and sizes (ranging from a few KBs to a few GBs), and integrate Samza and Ambry into a unified scalable media storage and processing environment.

5. REFERENCES

[1] Ambry. `http://www.github.com/linkedin/ambry`, (accessed Mar, 2016).

[2] HIPI: Hadoop image processing interface. `http://hipi.cs.virginia.edu/index.html`, accessed Mar, 2016.

[3] OpenCV: Open source computer vision. `http://www.opencv.org/`, accessed Mar, 2016.

[4] ShapeLogic. `http://www.shapelogic.org`, accessed Mar, 2016.

[5] Stormcv. `https://github.com/sensorstorm/StormCV`, accessed Mar, 2016.

[6] Transforming images into information. `http://4quant.com`, accessed Mar, 2016.

[7] YouTube statistics. `https://www.youtube.com/yt/press/en-GB/statistics.html`, accessed Mar, 2016.

[8] S. A. Noghabi, S. Subramanian, P. Narayanan, S. Narayanan, G. Holla, M. Zadeh, T. Li, I. Gupta, and R. H. Campbell. Ambry: Linkedin's scalable geo-distributed object store. In *Proceeding of the ACM Special Interest Group on Management of Data (SIGMOD)*, 2016.

[9] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel:

fault-tolerant stream processing at internet scale. *In Proceeding of the Very Large Data Bases Endowment (VLDB)*, 2013.

[10] A. Auradkar, C. Botev, S. Das, D. De Maagd, A. Feinberg, P. Ganti, L. Gao, B. Ghosh, K. Gopalakrishna, et al. Data infrastructure at LinkedIn. In *Proceeding of the IEEE International Conference on Data Engineering (ICDE)*, 2012.

[11] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in Haystack: Facebook's photo storage. In *Proceeding of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[12] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2), 2008.

[13] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceeding of the ACM SIGOPS Operating Systems Review (OSR)*, 2007.

[14] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceeding of the ACM SIGOPS Operating Systems Review (OSR)*, 2003.

[15] Hortonworks. Ozone: An object store in HDFS. `http://hortonworks.com/blog/ozone-object-store-hdfs/`, 2014 (accessed Mar, 2016).

[16] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. In *Proceeding of the ACM SIGOPS Operating Systems Review (OSR)*, number 2, 2010.

[17] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith. Andrew: A distributed personal computing environment. *Communications of the ACM (CACM)*, 29(3), 1986.

[18] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, et al. F4: Facebook's warm blob storage system. In *Proceeding of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[19] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Proceeding of IEEE International Conference on Data Mining Workshops (ICDMW)*, 2010.

[20] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network file system. In *Proceeding of the USENIX Summer Technical Conference*, 1985.

[21] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proceeding of the IEEE Mass Storage Systems and Technologies (MSST)*, 2010.

[22] A. Sozykin and T. Epanchintsev. MIPr - a framework for distributed image processing using Hadoop. In *Proceeding of the IEEE Application of Information and Communication Technologies (AICT)*, 2015.

[23] C. Sweeney, L. Liu, S. Arietta, and J. Lawrence. HIPI: a Hadoop image processing interface for image-based mapreduce tasks. *Chris. University of Virginia*, 2011.

[24] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. Storm@ twitter. In *Proceeding of the ACM Special Interest Group on Management of Data (SIGMOD)*, 2014.

[25] Twitter. Blobstore: Twitter's in-house photo storage system. `https://blog.twitter.com/2012/blobstore-twitter-s-in-house-photo-storage-system`, 2011 (accessed Mar, 2016).

[26] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceeding of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[27] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceeding of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2013.