**Title:** Lambda Architecture As A Means To Solve Big Data Problems

**Authors:** Justin Baraboo, Billy Capps, David Rodgers, Brendon O’Connor

**Abstract:**

Data is growing at an alarming rate which both good and bad. More data can lead to richer models; however, knowing what data should be trained on and what data shouldn’t is difficult. Especially as the volume of data is large and a problem in and of itself for categorizing and indexing. Classification needs to done in real time for many use cases, resulting in a small query time and reinforcing the need of modeling. In this paper review, we talk about the lambda architecture and how it serves as the driving force of many technologies and solutions to problems today for real time data analysis.

**Introduction:**

The data market is booming. According to IBM, 90% of the world’s data has been created in just the past two years alone, signaling both the strides in data production and storage (IBM). This growth of data has been a blessing and a curse; while more information allows us to make richer, better models too much data, especially data that doesn’t pertain to the task at hand, can obfuscate true relationships. Also, indexing, sorting, searching and transmitting the correct data has become challenging. As the data has grown tremendously and the time in which we need the output of the query decreases, clever methods of finding and sorting and consistent data are needed as quadratic and even n-logarithmic time complexities are too slow, for sequential processing. These problems lead to parallel processing techniques, both through task and data parallelism, and creating frameworks and architectures for parallelism.

Data itself is becoming less static and structured. 100 hours of video data are uploaded to Youtube every minute (Youtube). In one year, 459,000 years worth of video is streamed through Twitch services with 1.7 million broadcasters each month (Twitch). In order to process such data, real time analysis is needed as storing that much data is expensive especially as most of it could give very little use to the task at hand. So, being able to process the videos and no longer need them by training a model from them or being able store only the videos deemed important is critical.

To tackle this problem, we look at a lambda architecture that maximizes performance, availability, scalability, modifiability, portability, and usability. This architecture allows for streaming data as well as model processing in order to handle a task. The lambda architecture itself is simple in design, it consists of a batch processing unit (where we will use Spark), a speedy stream processing unit (where we will use Storm), and a merging unit to combine these results(where we will use Kafka) (Lambda). In this paper we detail the components of the lambda architecture, while talking about the mapreduce paradigm for solving problems in parallel, while also discussing the achievements and capabilities of parallel processing with a focus on image and video processing.

Before discussing each component of the lambda architecture we are interested in, it is imperative to talk about the mapreduce paradigm that allows Spark and Storm to ingest and parallelize the data. While there are many ways of achieving parallel computation and processing, Google’s mapreduce serves as a highly scalable, easy to implement solution for parallelism. The architecture consists of a master that controls many worker nodes which can either be on a local machine, a cluster, or distributed across machines, a grid. Data is split between the workers and operated on in parallel through mapping and reducing functions. The intermediate values of these functions are written onto local disk space (Google).

The main functions necessary are a mapping function, which maps data to key value pairs, and a reducing function, which acts on those key values pairs to solve a problem. Internally, a shuffling function is performed that swaps data so that all of the same keys move to the same partition. Hadoop HDFS also uses this mapreduce paradigm to solve problems in a parallel manner. However, each mapping and reduction step is written to a disk location, calling I/O operation to do so which are costly. To fix this, Spark was created.

**Body:**

Spark was created by UC Berkley as an extension to Hadoop’s map reduce framework. Spark’s key difference is that instead of writing to the disk with each intermediary result, such as from a map or reduce, it writes instead to a specialized distributed memory system each worker can access known as an RDD- resilient distributed dataset. Since disk I/O is a costly operation, this reduces the overhead time of map reduction greatly.

Spark is highly efficient when the data flow of a job is isn’t acyclic, that is the same data set is reused one or many times. Machine learning and many algorithms are fraught with non-acyclic data flow as parameter estimation for model generation, such as expectation maximization, learning, such as neural networks, or optimization, such as gradient descent, are used frequently. In these cases, data sets used are typically large and require a lot of time taken, so being able to reduce the time complexity by a factor or 100 or more is critical. Also, performing different, telescoping, or overlapping queries on a same data set is a common use case for industry and database needs. Spark performs well here as the subsequent queries can be performed on the already fetched data instead of needing to perform another disk I/O operation.

These strengths are truly shown in the spark ecosystem which includes an extensive machine learning library as well as SparkSQL, allowing users to easily make queries, working with SQL command structure.

The workflow architecture of Spark is fairly simple. Spark runs on top of Mesos, another UC Berkley invention, that handles the cluster aspect of map reduction and parallel processing. While YARN is used also used as a resource manager today, when the paper was written in 2010, it wasn’t fully evolved and used the main resource manager of map reduction in Hadoop until 2012 (Datanami).

The entry point for spark is through a spark context which works as a driver of the architecture. The driver program contacts the cluster manager which allocates resources and creates worker nodes. The worker nodes each contain an executor which contains tasks which they perform as well as a cache for data.

Spark’s defining feature is the RDD. An RDD is a distributed memory data set which can be accessed by the worker nodes. It is immutable as it can only be read from, not written to, this allows fast access to the data without having to perform lock mechanisms for consistency or having to deal with deadlocks. This is, however, highly restrictive on what operations can be performed on an RDD.

The creation of an RDD is easily done by a programmer. Text files and lists can be converted to RDDS. An RDD is also created when mapping is performed. Finally, RDDs can be created by changing their persistence. RDDs are not actually in the distributed memory until they have to be in order to perform a computation: this is a wise choice as if a user uses many commands at the start of the program to generate RDDs, they would have lots of memory space being wasted if they weren’t using them all, if RDDs were interpreted line by line and put into memory at creation.

Parallel operations on RDDs are also easy to do. Spark employs functions that allow “actions” on the RDDs: reduce, collect, foreach. These perform the reducing step in mapreduce paradigm. Finally, Spark allows variable sharing through the use of accumulators and broadcast variables. Broadcast variables allow a user to distribute a variable (such as a read only table) to workers, only once, instead of having to package it and transfer it with every call, thus saving time. Accumulators are used to “add” up some sort of count among all workers. In the simplest case of accumulating ints, it could just be a counter of something occurring. In current Spark, users can define their own accumulators in Scala allowing them to accumulate things other than ints.

RDDs are fault tolerant data structures. They achieve this by a concept called lineage. RDD creation and transformation can be thought of a chain of functional commands on one or more set(s), starting from disk and then on the in memory dataset. As such, if this functional chain is kept in order and the location of the original data, if at any point a node fails or data is corrupted, it can access the last current existing RDD and perform the functional chain on that data set until it regenerates the data. This is technically achieved by having each dataset object contain a pointer to its parent along with how its parent was transformed. This allows fault tolerance to be achieved without check pointing, a costly mechanism.

Spark fits into our lambda architecture as the batch processor. As it doesn’t have a streaming service, though can do micro batch jobs, and has a dedicated machine learning library, it makes sense for it server as the batch processor especially considering that Storm lends itself very well for streaming analysis though its architecture and framework.

Storm was developed at twitter to be able to handle real time processing of data. The design of this system was built around the following principles: scalable, resilient, extensible, efficient, easy to administer (Toshniwal). The system needed to be scalable in order to handle the massive amounts of data that twitter was receiving from its users. To achieve this, the system needed to be distributed across many commodity hardware machines and seamlessly pass data between these servers. Because the computers running the computation on this massive amount of data are prone to failures the storm system had to be resilient to failures at any step in the computational process as well. This ensured that any data that may have been in process are not lost and the system is able to easily recover and pick up where it started before the failure. The system also needed to be extensible because it was being used in the context of many different tools, so it had to be able to grow and accommodate any unforeseen changes that may occur down the line and easily integrate with them. The last two principles were more based on the practical realities. Because the system will need to provide timely insights, it needs to be time and resource efficient (to keep the cost of computation low). Finally it had to be easy to administer, realizing that real people need to be able to use this system. If Storm could only be implemented by a small specialized few, it would limit its ability to spread and not be utilized to its full potential. It is the collection of these design principles that set Storm apart from the other stream processing engines that were available.

The key concepts of storm involve topologies, spouts, bolts, and tuple streams. The stream processing is all done around tuples. All the data that flows through the Storm system is tuple data. The topology is the map which defines how the tuples flow through the Storm system. This map is later used by the system to direct the flow of tuples and allocate tasks to the available resources on the computational cluster. Each topology will specify at least one spout. The spout is the device that pulls in data streams to be processed. There can be multiple spouts in any given topology. From there the spouts pass the data to the bolts. These are the objects that process the tuples that are sent into the system. Whatever needs to be done to the tuples will be defined in each respective bolt. Spouts pass tuples to bolts and bolts pass tuples to other bolts. Typically there will be a bolt at some point in the system that outputs the data from Storm. This can take place in the form of a client or a database to be accessed later on.

Moving deeper down in the system, it is designed around being distributed. This means it integrates with distributed cloud clusters and passes the data across these servers to process different tasks. The “master” node is referred to as nimbus. This node is responsible for coordinating all of the tasks on the cluster. The “worker” nodes that perform the tasks are referred to at supervisor nodes (Toshniwal). This is fairly similar to the Spark architecture defined above. Each supervisor has a worker that executes the topology. The nimbus is able to coordinate tasks through the topology by “heartbeats”; the supervisor nodes during every specified interval of time (usually every few seconds) will send a signal to the nimbus node. These signals convey some information about the status of the corresponding worker node such as whether it currently is running a task or not. From this information and the topology, the nimbus is able to allocate whatever tasks need to be completed to the worker node that may not have any work. The nimbus sends that information to the supervisor which has tasks running to manage the worker node and starts any tasks that are requested and integrates those tasks within whatever else the worker node may also be working on. All of these messages are passed through the underlying Zookeeper server by which the storm system runs on. While this may seem unnecessary or even redundant, the added cost in communication time is made up for by the stability of the system. It is these Zookeeper servers that are able to store the state of the system and what needs to be executed. So whenever there are any failures in any part of the system, no information is lost. Any failure that occurs on one machine, the rest of the system will continue while that machine reboots and is able to communicate again with the rest of the system. This is a tremendous asset to storm’s resilient design principle.

The last mechanism that storm implements to make its system resilient in the face of common hardware failures is acker bolts. These acker bolts are able to keep track of each tuple that goes through the system. At the point of creation for each tuple, it has an id that is sent to the bolt that tracks it before and after each bolt. It is also able to track any tuples that are created by breaking a tuple into multiple tuples (Toshniwal). By doing this, it is able to verify when a tuple that enters the system has been completely processed. When this occurs it sends a notice to the tuple stream source that it can let go of the tuple. In the case of any failures, the bolt would notify the tuple source and pull the tuples that were lost from the failure and start again from where it left off. Through this system storm is able to guarantee that each tuple that enters the system is processed at least once. When this acker bolt is turned off through (likely due to performance concerns) it can only guarantee that the tuple is processed at most once. That means if a tuple enters the system it cannot guarantee that it won’t be lost at any point if one of the hardware systems is to to go down.

Storm fits into our lambda architecture as the fast streaming processor. It’s spout and bolt topology combined with tuple processing makes it very natural to handle streamed data. To finish the lambda architecture, we need a messaging system to combine the Storm and Spark outputs. We will use Kafka for this.

Kafka was originally developed as log processor by LinkedIn (Kreps). It processes logs in a real time system as a publisher subscriber system which lends itself to evolve into a much more robust data processor than just log processing. Kafka’s architecture consists of a consumer, producer, and broker. Data handling revolves around topics. The producer creates data that follow under a certain topic. This is handled by the brokers, a set of servers to store the data until consumption, under that topic. A consumer subscribes to topics and pulls them from the broker. Kafka allows for very efficient data transfer by removing a lot of the overhead from caching messages. Kafka is also a stateless broker as the brokers do not keep track of how much a consumer has consumed: the consumer does. Data is removed after a certain time period and consumers consume through Kafka by offsetting their looks in the “log” for new data. Kafka also guarantees at least once delivery as defined above, making it fault tolerant without adding too much complexity or time burden (Kreps). Underscoring these ideas, Kafka becomes a fault tolerant, efficient, scalable, and fast messaging system.

Kafka forms the broker between Spark and Storm in our lambda architecture by allowing them to publish and consume different topics as well as being able to query and talk to other outside components to push or pull data as needed to run our lambda architecture. The use of this architecture allows us to implement a fast, rich modeling and classification and tasks in a parallel manner allowing many application to thrive such as image and video analysis.

A lambda architecture is used within the paper “A Deep Intelligence Framework for Online Video Processing” to process online videos . In this paper, they use Hadoop as their batch processing unit and Storm for their streaming processing unit with a few different services to act as the merger (but none of them Kafka). Here, they wanted to train a deep convolutional neural network in Hadoop offline and stream videos and classify things like vehicles or traffic jams online through Storm(Zhang). Since each component of their architecture contributes to fault tolerance, scalability, modifiability, and portability, their entire architecture is as well, mostly through Storm being those and being the online streaming unit. Since they are performing neural network training on data offline through Hadoop, it may be faster to use Spark as neural network training uses a lot of back propagation and iterations on the same network and data to reduce error when training the model. In Hadoop, this would correspond to many I/O writes. Another use of neural networks and lambda architecture is cloud computing, vision, and automation.

Automation typically involves workflows designed for a specific task or goal in an isolated environment that is well-organized and specific to the task at hand. While these solutions are accurate and efficient, they are only adaptive in these very specific scenarios. The future of automation relies on a much more flexible environment, where the setup is less structured and can work with different people and/or systems. In terms of data, this means being reliant on flexible computational solutions, both on a system and algorithmic level. The use of a cloud-based computation offers this flexibility, as cloud-based computing involves shared, customizable resources from a hardware level to a software or even platform level. A more important implication of this is the rise of Big Data. With the sharing of resources and information, cloud-based computing can have access to global-scale libraries and information bases, which can be used to create flexible automation environments for robots. Together with Big Data, cloud automation can provide opportunities such as improved perception, faster planning, accurate modeling, lifelong learning, large-scale systems, sophisticated robots, new ways of interacting with people. Thus, a lambda-like cloud architecture for a synergistic robot, one that splits computation between remote cloud and the robot, appears advantageous for cloud automation.

The main goals of an effective synergistic model includes computational efficiency, communication overhead, safety and reactivity to changes, and path quality (Bekris et al). How a robot creates its roadmap gives insight to effectiveness of its model. A roadmap can be computed locally or it can pre-computed, such as a heuristic or algorithm. Overall, successful roadmaps have properties that are cloud-based and relies on both local and global computations. One type of roadmap is the probabilistic roadmap (PRM), which is a motion planning algorithm in robotics that solves the problem of determining a path between the robot (i.e., the starting location) and an endpoint all while avoiding collisions.There are also large dense roadmaps that are beneficial but a tradeoff has to be achieved between query resolution time and path quality. There is the Incremental Roadmap Spanner (IRS), which is similar to PRM except that it has only a subset of the PRM’s edges which usually has a lower query resolution time and less degradation in path quality. Then there is the Sparse Roadmap Spanner (SPARS), which is better than IRS in terms of roadmap size and online query resolution time. The Baxter arm, which is a robotic arm that is used to transfer bottles to a different location, is used to evaluate PRM, IRS and SPARS (Bekris et al). The researchers also played with difficulty by varying the location and complexity of the space of endpoint of the bottle, and providing new structures that the robot learns for the first time in the experiment. The results show that SPARS is the fastest in most cases in query resolution (which is due to the sparsity of its roadmap), while PRM generates roadmaps with the best path quality. All three methods can achieve the easy environment, but SPARS does not do too well in difficult tasks (Bekris et al). All things considered, however, the fact that SPARS can complete hard tasks with the sparsest roadmap is quite a feat.

Overall, the need to balance sparsity and adaptability is important for cloud automation. More data to compute in real-time might improve accuracy and quality, but the time to process will be too long; on the other hand, having few data points to work with might give you faster processing speed, but not necessarily high reliability that a cloud-based synergistic robot would need. It appears the components of the lambda architecture are implemented within this design, as cloud information must be streamed to the robot to help the robot learn about the roadmap on a global scale, while it locally batch process information it takes in as to update its model and learn more efficiently. Indeed, the researchers conclude that robots what actively explore and update their roadmaps in real-time will be most useful, and perhaps with more testing, this might achieve the balance between sparsity and adaptability that is needed to create a fast and reliable synergistic robot.

Another important model that handles the flexibility and adaptability of learning with large datasets is a model that uses unsupervised learning of video sequences using LSTM. LSTM stands for Long Short Term Memory, and it is often used in recurrent neural networks (RNNs) to assist with learning speech recognition, machine translation and caption generation of images (Srivastava et al). In this study, the researchers investigated how they could create a model or representation that could remember important information in a video and use it to predict what the next few frames of the video would be. An example would be a ball bouncing in which the model can track the motion of the ball and use that knowledge to predict where the ball will go. The researchers point out that supervised learning is useful for visual representations, but for a video, they decided to go from an unsupervised learning approach. Their reasoning is that it would be better to learn on a dataset of unlabelled videos to inform a supervised learning model, mainly because videos are too complex and would require a lot of labelled training data to be able to find predictive/useful features for supervised learning to be efficient (i.e., low dimensionality). Videos are structured in a very predictable spatial and temporal way, which can lend itself to unsupervised learning (Srivastava et al). Combined with RNNs that use LSTM, an unsupervised learning approach captures a much more realistic learning environment as it involves learning in real-time vs a lab-setting environment in which data is already pre-processed, labelled and available to be used as training data.

The main framework of their model is a sequence to sequence learning approach, where one RNN encodes sequence into a fixed length representation, another decodes a sequence from that representation. With this framework, they can learn about sequence of images (i.e., a video). With this framework, they can achieve three main tasks: one to predict the same sequence as the input, one to predict future frames given the input, and another to do both which would involve two decoders. The inputs are image patches and high-level “percepts”, which are the states of the rectified linear hidden units states that are found in the last and/or second-to-last layer of the neural network. The researchers designed the encoder-decoder RNN architecture with adaptability in mind so it could be compatible with any differentiable loss function (Srivastava et al).

Two video datasets were used to evaluate the LSTM, in which one had 101 action categories and the other had 51. The researchers compared a single frame classifier to a LSTM classifier and a LSTM classifier with pretraining. Overall, features learned by unsupervised model helps improve performance slightly on supervised tasks. Specifically, a LSTM classifier performs better than the single frame classifier in its ability to classify different action categories, and while not surprising, the LSTM classifier perform the best (Srivastava et al). It is important to note that on the whole, the more training examples given, the better each of the classifiers performed, but this effect seemed to diminish as the number of training examples past 100. This implies that the ability of having an unsupervised learning model as part of an overall learning model is crucial, as it is not realistic to expect to have available labelled data in the real-world in real-time.

The results from the Unsupervised Learning with Long Short Term Memory underscores the importance of adaptability of a defined model. From a lambda architecture perspective, data is streamed in as a video. The three main components of the model represents the main streaming and batch processing nature of lambda architecture: autoencoder and future predictor are each an example of batch processing in which part of a stream of video is analyzed, while the composite model has more stream-like approach in which the entire set (both the reconstruction of the input and the prediction of what comes next) is taken into account. The model even exhibits the flexibility that is required for big data - data that is both similar and expected (supervised learning), and data that is equally relevant but whose value does not explicitly fit into a given model, which is computed with the unsupervised LSTM classifier. While the researchers went out of their way from doing a simple identity mapping, they also emphasized the fact that neither supervised nor unsupervised learning had the best performance in classifying action categories, but the combination together, the ability to learn by given example and to learn in real-time, is what gave the best performance.

WiseReplica is a new approach to handling the increasing demands of video on demand services that are becoming popular (such as YouTube or Netflix) (Silvestre). The increased demand for these systems have created a need to handle the video data efficiently in order to keep the product quality high and fast. WiseReplica was designed in order to fix the inefficiencies of this problem. It was able to reduce the need for replication by a twofold order of magnitude and increase the bitrate by 85%. The main focus of WiseReplica, though, is to increase quality by decreasing the need to replicate videos as they are requested. One of the key insights of this system is to store videos that are frequently request in order to reduce the time it would take to constantly replicate them.

WiseReplica is able to decrease the replication by intelligently anticipating the needs of the users. It does this by taking in data on the various requests made of the system and apply machine learning algorithms to be trained on the request data. Once the system has been trained it is able to make better adaptive decisions on which videos to store in order to reduce the need for replication. Without this technique it is hard to statically determine the demand for various videos because it is constantly influx and decisions have to be made in real time. On top of that WiseReplica uses a storage domain to distribute videos. Each storage domain has two roles. The first is the coordinator whose job it is to figure out the scheduling of videos. The second is the peer that sends the actual video to the customers. Through this two fold system coupled with accurate prediction measures WiseReplica was able to produce the transformational results as stated above.

WiseReplica’s role in the lambda architecture is assisting in more intelligent use of resources. It plays a role in a decision maker to help reduce the cost of resources when using video data and selecting which data to store for easy access. WiseReplica is able to decrease the burden on the system by eliminating repeated actions for requests on the data.

**Conclusion**

The lambda architecture is an extremely powerful framework for allowing scalability, efficiency, portability, and easy to manage and administer tools. In this paper, we discussed how Storm, Spark, and Kafka can work together to form a lambda architecture and their strengths of doing so. Spark is a faster implementation of Hadoop’s HDFS mapreduce by writing and reading from a distributed memory system, RDDs, especially in the case of parameter estimation and model training. These strengths allow it to do machine learning and batch processing offline. Storm’s topology of bolts and spouts along with tuple processing allows it easily stream in data and process it, begging for it to be the fast streaming component for real time analysis. Kafka is a publisher subscriber system that is extremely scalable and fault tolerant with its log system and brokers and topics. It allows Spark and Storm to communicate with each other as well as databases and feed information into them. In general, the lambda architecture fuels many real time parallel designs. The idea of combining the streaming advantages of the cloud with local processing in robots provides a new realm of cloud automation, which can not only better robotic roadmap calculations, but can provide a platform on which to expand Big Data opportunities involving larger scale robotic systems that interact with people on a real-time basis. WiseReplica approach helps to make the use of video data more efficient and reduce the stress on the system. This reduces overall demand constraints for transferring data through the lambda architecture. The unsupervised learning approach with Long Short Term Memory meets the common middle ground of both stream and batch processing as well as maintaining the fine balance of learning based on a given model and learning in real time. The lambda architecture thus provides the basic, but easily applicable framework of flexible learning in real-time.

**Future Research Directions**

Future research directions are endless. Combining batch processing to train a neural network or a complicated model and then using it in real time allows the best of both worlds: good classification and in a short amount of time. Many problems, especially image analysis and retrieval, require complicated modeling or aggregation schemes but fast and accurate classification. The architecture itself is also modular, allowing more components to be added (or removed) as they merely talk via the merger. So instead of having one batch layer, we could have many with only partial data sets to bias their modeling. Medical diagnosis or assistance could benefit from lambda architecture and parallel computing by quickly surveying the existing data and recommending a course of action to the doctor. Also, as seen from the online video processing paper, the lambda architecture can easily be nested with a greater bigger design. Even though the growth of data is incredibly exponential and shows no signs of slowing down, creative solutions and designs allow its use for better modeling, better compression, and better classification despite the torrent of useless data while being fault tolerant, scalable, and usable.

**References**

*Cloud Automation*

Bekris K, Shome R, Krontiris A, Dobson A. "Cloud Automation: Precomputing Roadmaps for Flexible Manipulation", *IEEE Robotics & Automation Magazine* Vol 22: Issue 2, 2015.

*Datanami*

Alex Woodle. “From Elephants to Spiders: The History of Hadoop”.

<https://www.datanami.com/2015/04/15/from-spiders-to-elephants-the-history-of-hadoop/>

*Google*

Dean J, Ghemawat S. "MapReduce: simplified data processing on large clusters". *Communications of the ACM Magazine - 50th anniversary issue: 1958-2008*, Vol 51: Issue 1, Jan 2008.

*IBM*

"What is big data?" *IBM - Bringing Big Data to the Enterprise.* URL: <https://www-01.ibm.com/software/data/bigdata/what-is-big-data.html>

*Kafka*

Kreps J, Narkhede N, Rao J. "Kafka: a Distributed Messaging System for Log Processing". *NetDB Workshop*, 2011.

*Lambda*

Bijnens N. "A real-time architecture using Hadoop and Storm". *DataCrunchers*, Dec 2013. URL: <http://lambda-architecture.net/architecture/2013-12-11-a-real-time-architecture-using-hadoop-and-storm-devoxx>

*Unsupervised Learning/LSTM*

Srivastava N, Mansimov E, Salakhutdinov R. "Unsupervised Learning of Video Representations using LSTMs", *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pp 843-852, 2015.

“Boosting Streaming Video Delivery with WiseReplica”

Silvestre S, Buffoni D, Pires K, Monnet S, Sens P. "Boosting Streaming Video Delivery with WiseReplica". *Transactions on Large-Scale Data-and Knowledge-Centered Systems*, 34-58.

*Storm*

Toshniwa A, Taneja S, Shukla A, Ramasamy K, Patel JM, Kulkarni S, Jackson J, Gade K, Fu M, Donham J, Bhagat N, Mittal S, Ryaboy D. "Storm @Twitter". *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pp 147-156, 2014.

*Twitch*

“Welcome Home the 2015 Retrospective” URL:

URL: <https://www.twitch.tv/year/2015>

*YouTube*

“Statistics - YouTube”. *YouTube* URL: <https://www.youtube.com/yt/press/en-GB/statistics.html>

*Deep-Intelligence Framework*

Zhang W, Xu L, Li Z, Lu Q. "A Deep-Intelligence Framework for Online Video Processing". *IEEE Computer Society*, pp 44-51, Mar 2016.