Lambda Architecture – Realtime Data Processing

*Dr. Yuvraj Kumar – Ph.D. in Artificial Intelligence
*The faculty of Quantum Computing and Artificial Intelligence
*Zukovsky State University, Russia

Abstract - Data has evolved immensely in recent years, in type, volume and velocity. There are several frameworks to handle the big data applications. The project focuses on the Lambda Architecture proposed by Marz and its application to obtain real-time data processing. The architecture is a solution that unites the benefits of the batch and stream processing techniques. Data can be historically processed with high precision and involved algorithms without loss of short-term information, alerts and insights. Lambda Architecture has an ability to serve a wide range of use cases and workloads that withstands hardware and human mistakes. The layered architecture enhances loose coupling and flexibility in the system. This a huge benefit that allows understanding the trade-offs and application of various tools and technologies across the layers. There has been an advancement in the approach of building the LA due to improvements in the underlying tools. The project demonstrates a simplified architecture for the LA that is maintainable.

Index terms – ***Lambda Architecture (LA), Batch Processing, Stream Processing, Real-time Data Processing***

I. INTRODUCTION

With tremendous rate of growth in the amounts of data, there have been innovations both in storage and processing of big data. According to Dough Laney, Big data can be thought of as an increase in the 3 V's, i.e. Volume, Variety and Velocity. Due to sources such as IoT sensors, twitter feeds, application logs and database state changes, there has been an inflow of streams of data to store and process. These streams are a flow of continuous and unbounded data that demand near real-time processing of data. The field of data analytics is growing immensely to draw valuable insights from big chunks of raw data. In order to compute information in a data system, processing frameworks and processing engines are essential. The traditional relational database seems to show limitations when exposed to the colossal chunks of unstructured data. There is a need to decouple compute from storage. The processing frameworks can be categorized into three frameworks –batch processing, stream data processing and hybrid data processing frameworks. Traditional batch processing of data gives good results but with a high latency. Hadoop is a scalable and fault-tolerant framework that includes Map Reduce for computational processing. [1][2] Map Reduce jobs are run in batches to give results which are accurate and highly available. The downside of Map Reduce is its high latency, which does not make it a good choice for real-time data processing. In order to achieve results in real-time with low-latency, a good solution is to use Apache Kafka coupled with Apache Spark. This streaming model does wonders in high availability and low latency but might suffer in terms of accuracy. In most of the scenarios, use cases demand both fast results and deep processing of data.[3] This project is focused towards Lambda Architecture that unifies batch and stream processing. Many tech companies such as Twitter, LinkedIn, Netflix and Amazon use this architecture to solve multiple business requirements. The LA architecture aims to meet the needs of a robust system that is scalable and fault-tolerant against hardware failures and human mistakes. On the other hand, the LA creates an overhead to maintain a lot of moving parts in the architecture and duplicate code frameworks.[4] The project is structured into different sections. The Lambda Architecture emerges as a solution that consists of three different layers. The LA is an amalgamation of numerous tools and technologies. The LA fits in many use cases and has applications across various domain.

II. BACKGROUND

Traditional batch processing saw a shift in 2004, when Google came up with Hadoop MapReduce for big data processing [1]. MapReduce is a scalable and efficient model that processes large amounts of data in batches. The idea behind the MapReduce framework is that the collected data is stored over a certain period before it is processed. The execution time of a batched MapReduce job is dependent on the computational power of the system and the overall size of the data being processed. That is why, large scale processing is performed on an hourly, daily or weekly basis in the industry as per the use case. MapReduce is widely used for data analytics with its batch data processing approach but tends to fall short when immediate results prerequired recent times, there has been a need to process and analyze data at speed. There is a necessity to gain insights quickly after the event has happened as the value diminishes with time. Online retail, social media, stock markets and intrusion detection systems rely heavily on instantaneous analysis within milliseconds or seconds. According to [5], Real-time data processing combines data capturing, data processing and data exploration in a very fast and prompt manner. However, MapReduce was never built for this purpose, thereby, leading to the innovation of stream processing systems. Unlike batch processing, the data fed to a stream processing system is unbounded and in motion. This can be time series data or generated from user's web activity, applications logs, or IoT sensors, and must be pipelined into a stream processing engine such as Apache Spark [9] or Apache Storm [11]. These engines have the capability to compute analysis that can be further displayed as real-time results on a dashboard.
Stream processing is an excellent solution for real-time data processing, but the results can sometimes be approximate. As data arrives in the form of streams, the processing engine only knows about the current data and is unaware of the dataset. Batch processing frameworks operate over the entire dataset in a parallel and exhaustive approach to ensure the correctness of the result. Stream processing fails to achieve the same accuracy as that of batch processing systems. Batch and stream processing were considered diametrical paradigms of big data architecture until 2013, when Nathan Marz founded the Lambda Architecture (LA) [3] describes how LA addressed the need of unifying the benefits of batch and stream processing models. According to Marz, the following are the essential requirements of the LA:
• Fault-tolerant and robust enough to withstand code failures and human errors
• The layers must be horizontally scalable.
• Low latency in order to achieve real-time results
• Easy to maintain and debug.

III. LAMBDA ARCHITECTURE

The proposed architecture is demonstrated in Fig. 1. The incoming data is duplicated and fed to the batch and speed layer for computation.

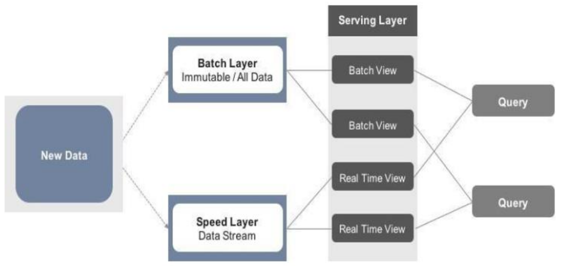


Fig. 1. Lambda Architecture [2]

[3] discusses in detail about the three layers in the LA. A subset of properties necessary for large-scale distributed big data architectures is satisfied by each layer in the LA. A highly reliable, fault-tolerant and low latency architecture is developed using multiple big data frameworks and technologies that scale out in conjunction across the layers.

A) BATCH LAYER

The crux of the LA is the master dataset. The master dataset constantly receives new data in an append-only fashion. This approach is highly desirable to maintain the immutability of the data. In the book [3], Marz stresses on the importance of immutable datasets. The overall purpose is to prepare for human or system errors and allow reprocessing. As values are overridden in a mutable data model, the immutability principle prevents loss of data. Secondly, the immutable data model supports simplification due to the absence of indexing of data. The master dataset in the batch layer is ever growing and is the detailed source of data in the architecture. The master dataset permits random read feature on the historical data. The batch layer prefers re-computation algorithms over incremental algorithms. The problem with incremental algorithms is the failure to address the challenges faced by human mistakes. The re-computational nature of the batch layer creates simple batch views as the complexity is addressed during precomputation. Additionally, the responsibility of the batch layer is to historically process the data with high accuracy. Machine learning algorithms take time to train the model and give better results over time. Such naturally exhaustive and time-consuming tasks are processed inside the batch layer. In the Hadoop framework, the master dataset is persisted in the Hadoop File System (HDFS) [6]. HDFS is distributed and fault-tolerant and follows an append only approach to fulfill the needs of the batch layer of the LA. Batch processing is performed with the use of MapReduce jobs than run at constant intervals and calculate batch views over the entire data spread out in HDFS. The problem with the batch layer is high-latency. The batch jobs must be run over the entire master dataset and are time consuming. For example, there might be some MapReduce. jobs that are run after every two hours. These jobs can process data that can be relatively old as they cannot keep up with the inflow of stream data. This is a serious limitation for real-time data processing. To overcome this limitation, the speed layer is very significant.

B) SPEED LAYER

[3] and [5] state that real-time data processing is realized because of the presence of the speed layer. The data streams are processed in real-time without the consideration of completeness or fix-ups. The speed layer achieves up-to-date query results and compensates for the high-latency of the batch layer. The purpose of this layer is to fill in the gap caused by the time-consuming batch layer. In order to create real-time views of the most recent data, this layer sacrifices throughput and decreases latency substantially. The real-time views are generated immediately after the data is received but are not as complete or precise as the batch layer. The idea behind this design is that the accurate results of the batch layer override the real-time views, once they arrive.
The separation of roles in the different layers account for the beauty of the LA. As mentioned earlier, the batch layer participates in a resource intensive operation by running over the entire master dataset. Therefore, the speed layer must incorporate a different approach to meet the low-latency requirements. In contrast to the re-computation approach of batch layer, the speed layer adopts incremental computation. The incremental computation is more complex, but the data handled in the speed layer is vastly smaller and the views are transient. A random-read/random-write methodology is used to re-use and update the previous views. There is a demonstration of the incremental computational strategy in the Fig. 2.

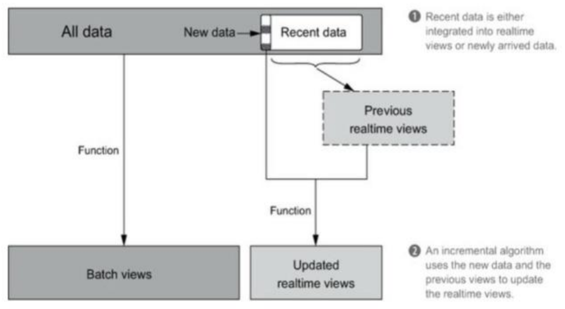

All data
New data
Recent data
1 Recent data is either integrated into realtime views or newly arrived data.
Previous realtime views
Function
Function
Batch views
Updated realtime views
2 An incremental algorithm uses the new data and the previous views to update the realtime views.

Fig. 2. Incremental Computation Strategy [3]

C) SERVICE LAYER

The serving layer is responsible to store the outputs from the batch and the speed layer. [3] An arrangement of flat records with pre-computed views are obtained as results from the batch layer. These pre-computed batch views are indexed in this layer for faster retrieval. This layer provides random reads and batch upgrades due to static batch perspectives. According to [3], whenever the LA is queried, the serving layer merges the batch and real-time views and outputs a result. The merged views can be displayed on a dashboard or used to create reports. Therefore, the LA combines the results from a data-intensive, yet accurate batch layer, and a prompt speed layer as per the required use case.

IV. TOOLS AND TECHNOLOGIES

The LA is a generic purpose architecture that allows a choice between multiple technologies. Each layer has a unique responsibility in the LA and requires a specific technology. Here is a list of a few technologies used across this architecture:

A) APACHE KAFKA

According to [7], Apache Kafka is a distributed pub-sub/messaging queue used to build real-time streaming data pipelines. A topic is used to store the stream of records inside Kafka. A publisher pushes messages into the topics and a consumer subscribes to the topic. Due to the fact that Kafka is a multi-consumer queue, the messages can be rewound and replayed in case of a point of failure. There is a configurable retention period to persist all published records irrespective of their consumption. The data generated from user website activity, applications logs, IoT sensors can be ingested into Apache Kafka. [10] shows how it is the responsibility of Apache Kafka to duplicate the data and send each copy to the batch and the speed layer respectively.

B) APACHE HADOOP

[2] defines Apache Hadoop as a distributed software platform for managing big data across clusters. The idea behind Hadoop was instead of bringing data towards compute, bring compute to the data. The Hadoop framework can be categorized into storage and compute models known as Hadoop Distributed File System and MapReduce respectively.

C) HDFS

HDFS is a scalable, reliable and fault-tolerant file system that stores huge quantities of data. HDFS is the most used technology for storage in the batch layer. [5] The immutable, append-only master dataset is dumped inside a resilient HDFS.

D) MAPREDUCE

According to [1] and [2], MapReduce is a programming paradigm that manipulates key-value pairs to write computations in map and reduce functions. In the map phase, the individual chunks of data generated through splits of input-data are processed in parallel. The output from the map phase is then sorted and forwarded to the reduce stage of the framework. The MapReduce framework runs over the master dataset and performs the precomputation required in the batch layer. Hadoop also includes Hive and Pig, which are high level abstractions that later translate to MapReduce jobs.

E) APACHE SPARK

As discussed in [9] and [10], the process of reading and writing to a disk in MapReduce is slow. Apache Spark is an alternative to MapReduce and runs up to 100 times faster due to in-memory processing. Spark works on the entire data as opposed to MapReduce, which runs in stages. Resilient Distributed Datasets (RDD) is the primary data structure of Spark that is a sharable object across jobs representing the state of memory. Spark is a polyglot and can run stand-alone or on Apache Mesos, Kubernetes, or in the cloud. Spark supports multiple features such as batch processing, stream processing, graph processing, interactive analysis and machine learning.

F) SPARK STREAMING

Spark streaming is an extension of core Spark API to enable processing of live data streams. The input data streams are divided into micro batches by Spark streaming and then final streams of results in batches are processed by the Spark Engine. Fig. 3. illustrates the same.

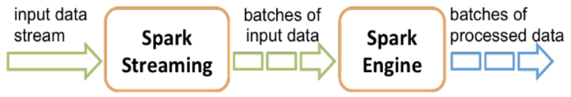


Fig. 3. Spark Streaming [10]

The live stream data can be ingested from a source such as Apache Kafka and can be analyzed using various functions provided by Spark. The processed data can be dumped out to databases, file systems, live dashboards or be used to apply machine learning. A stream of continuous data is represented with DStreams (discretized streams) and is basically a sequence of RDDs. The speed layer can be implemented using Spark Streaming for low latency requirements.

G) APACHE STORM

Apache Storm is the basis of the speed layer in the LA suggested by Nathan Marz [3].Instead of micro-batching the streams, Storm relies on a one-at-a-time stream processing approach. Due to this, the results have lower latency compared to Spark Streaming. [11] Storm is dependent on a concept called topologies which is equivalent to MapReduce jobs in Hadoop. A topology is a network of spouts and bolts. A spout is considered as the source of a stream, whereas the bolt performs some action on the stream.

H) APACHE HBASE

[13] Apache HBase is a non-relational distributed database that is a great option for the serving layer. It is a core component of the Hadoop Framework that handles large scale data. Due to extremely good read and write performance, the batch and real-time views can be stored and read in real-time. It fulfills the responsibility of exposing the views created by both the layers in order to service incoming queries.

I) APACHE CASSANDRA

Apache Cassandra is heavily used in the LA to store the real-time views of the speed layer. It is the preferred technology proposed by [3] to store the output from stream processing technologies. Along with that, it supports high read performance to output reports. Cassandra is a distributed database and has the ability to perform, scale and provide high availability with no single point of failure. [12] The architecture includes a masterless ring that is simple, easy to setup and maintain. Instead of maintaining a separate database in the serving layer like HBase or ElephantDB, Cassandra can be used to fulfill the same purpose. This reduces a lot of overhead and complexity.

V. APPLICATIONS OF LAMBDA ARCHITECTURE

Lambda architecture can be considered as almost real-time data processing architecture. It can withstand the faults as well as allow scalability. It uses the functions of batch layer and stream layer and keeps adding new data to the main storage while ensuring that the existing data will remain intact. In order to meet the quality of service standards, companies like Twitter, Netflix, and LinkedIn are using this architecture. Online advertisers use data enrichment to combine historical customer data with live customer behavior data and deliver more personalized and targeted ads in real-time and in context with what customers are doing. LA is also applied to detect and realize unusual behaviors that could indicate a potentially serious problem within the system. Financial institutions rely on trigger events to detect fraudulent transactions and stop fraud in its tracks. Hospitals also use triggers to detect potentially dangerous health changes while monitoring patient vital signs, sending automatic alerts to the right caregivers who can then take immediate and appropriate action. Twitter APIs can be called to process large feeds through the lambda architecture pipeline and sentiment analysis can be performed. Additionally, complex networks can be protected from intrusion detection by tracking the failure of a node and avoiding future issues.

As discussed in [14], MapR uses the Lambda architecture for online alerting in order to minimize the idle transports. The architecture used by MapR is demonstrated in Fig. 4.

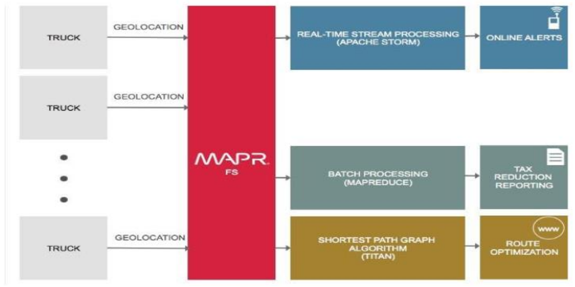


Fig. 4. Lambda Architecture at MapR [14]

In [15], a software development company named Talentica uses the LA for mobile ad campaign management. In a mobile ad campaign management, a processing challenge is faced due to high volumes and velocities of data. If real-time views are not generated, the campaign might fail to deliver and incur heavy losses in terms of revenue and missed opportunities. The company faced a situation where over 500 GB of data and 200k messages/sec were generated. The company needed a unified approach including both batch and stream and hence, the architecture demonstrated in Fig. 5. was established.

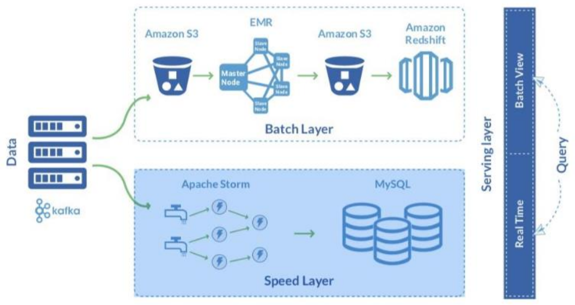


Fig. 5. Lambda Architecture at Talentica [15]

The LA turned out to be a cost effective and scalable solution to meet the demanding requirements of the mobile ad campaign.

[16] provides detailed information about how a LA was built on AWS to analyze customer behavior and recommend content by a software engineer who works at SmartNews.

SmartNews aggregates outputs from machine learning algorithms with data streams to gather user feedback in near real-time. The LA helps them achieve the best stories on the web with low-latency and high performance. The architecture followed at SmartNews can be seen in Fig.6.

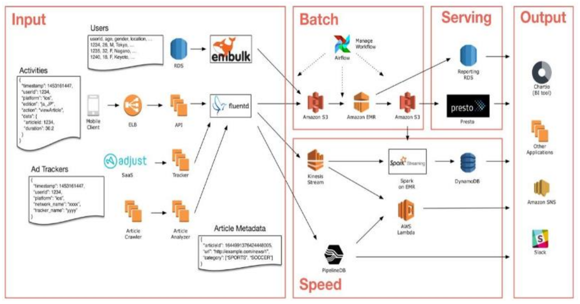


Fig. 6. Lambda Architecture at SmartNews[16]

Another use case of LA can be seen incorporated by the tech giant, Walmart Labs [17]. Walmart is a data driven company and produces business and product decisions based on the analysis of data. The use case focuses on click stream analysis in order to find the unique visitors, orders, units, revenues, site error rates. During peak hours, the pipeline holds as much as 250k events per second. The software engineers at Walmart devised a LA solution as illustrated in Fig. 7. for better productivity and results.

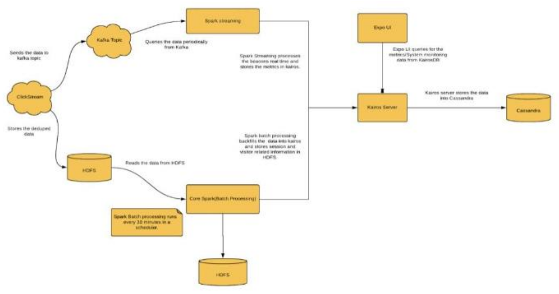

Spark streaming
Kafka Topic
Queries the data periodically from Kafka
Expo UI
Sends the data to kafka topic
Spark Streaming processes the beacons real time and stores the metrics in kairos.
Expo UI queries for the metrics/System monitoring data from KairosDB
ClickStream
Kairos Server
Kairos server stores the data into Cassandra
Cassandra
Stores the deduped data
Spark batch processing backfills the data into kairos and stores session and visitor related information in HDFS.
HDFS
Reads the data from HDFS
Core Spark(Batch Processing)
Spark Batch processing runs every 30 minutes in a scheduler.
HDFS

Fig. 7. Lambda Architecture at Walmart Labs [17]

This framework is a simplified version of LA because of the Spark implementation in the batch and speed layer. As discussed earlier, Spark provides APIs for both batch and Spark streaming. Due to this, there is a considerable decrease in the complexity of the code base. There is no need to maintain separate code frameworks as Spark enables code reuse.

VI. LIMITATIONS OF THE TRADITIONAL LAMBDAARCHITECTURE

Initially, when the LA was proposed, the batch layer was meant to be a combination of Hadoop File System and MapReduce and the stream layer was realized using Apache Storm. Also, the serving layer was a combination of two independent databases i.e. ElephantDB and Cassandra. Inherently, this model had a lot of implementation and maintenance issues. [4] Developers were required to possess a good understanding of two different systems and there was a steep learning curve. Additionally, creating a unified solution was possible but resulted a lot of merge issues, debugging problems and operational complexity. The incoming data needs to be fed to both the batch and the speed layer of the LA. It is very important to preserve the ordering of events of the input data to obtain thorough results. The process of duplicating streams of data and passing it over to two separate consumers can be troublesome and creates operational overhead. The LA does not always live up to the expectation and many industries use full batch processing system or a stream processing system to meet their use case.

VII. A PROPOSED SOLUTION

As understood, the LA can lead to code complexity, debugging and maintenance problems if not implemented in the right manner. A proper analysis and understanding of existing tools helped me realize that the LA can be implemented using a technology stack comprising of Apache Kafka, Apache Spark and Apache Cassandra. There are multiple ways of building a LA. In this approach, I will try one of them. I will use Apache Zeppelin to display the results observed in the serving layer.

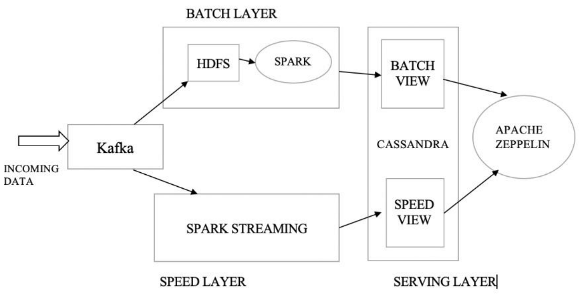
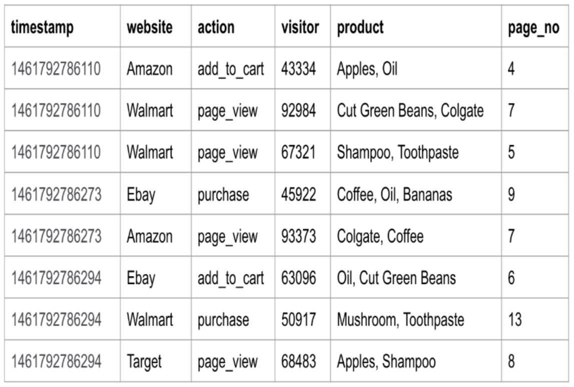


Fig. 8. Proposed Lambda Architecture

A) DATASET

As discussed earlier, LA handles fast incoming data from sources such as IoT sensors, web application clickstreams, application logs etc. In our case, a log producing continuous flow of clickstream is chosen as a dataset. The dataset resembles an online retail web application and the clickstream events. Basically, this is a simulation of streams in the form of logs. Kafka producer will publish this data on the broker which will be used by the downstream systems, i.e. the batch and speed layers. The log will be demonstrated in the following format:



timestamp	website	action	visitor	product	page_no
1461792786110	Amazon	add_to_cart	43334	Apples, Oil	4
1461792786110	Walmart	page_view	92984	Cut Green Beans, Colgate	7
1461792786110	Walmart	page_view	67321	Shampoo, Toothpaste	5
1461792786273	Ebay	purchase	45922	Coffee, Oil, Bananas	9
1461792786273	Amazon	page_view	93373	Colgate, Coffee	7
1461792786294	Ebay	add_to_cart	63096	Oil, Cut Green Beans	6
1461792786294	Walmart	purchase	50917	Mushroom, Toothpaste	13
1461792786294	Target	page_view	68483	Apples, Shampoo	8

Fig. 9. DataSet

B) APACHE KAFKA – DATA INGESTION

Kafka is a distributed streaming platform with the following capabilities:-Publish and subscribe to streams of records, similar to a message queue or enterprise messaging system.-Store streams of records in a fault-tolerant durable way.-Process streams of records as they occur. Kafka is used to build ingestion pipelines for scalable data processing systems and very well fits our real-time data processing use case. A brief introduction to the architecture and terminologies used in Kafka:

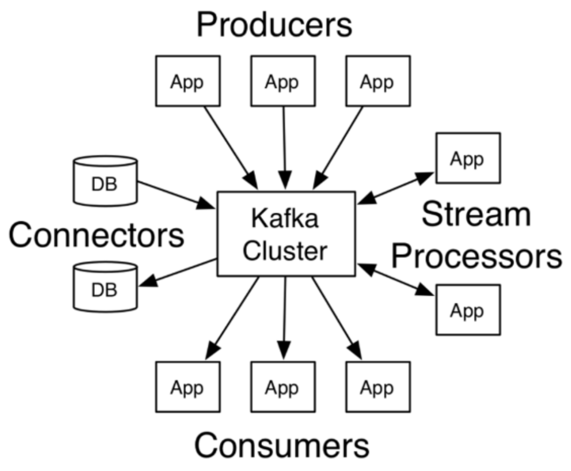


Fig. 10.Apache Kafka [7]

Kafka acts as a publisher-subscriber model and hence benefits from the loose coupling between the producers and the consumers. A Kafka cluster contains brokers to store the published messages from a producer. The consumers subscribe to the broker and receive the messages in order. The asynchronous nature of Kafka really helps out in building scalable architectures.

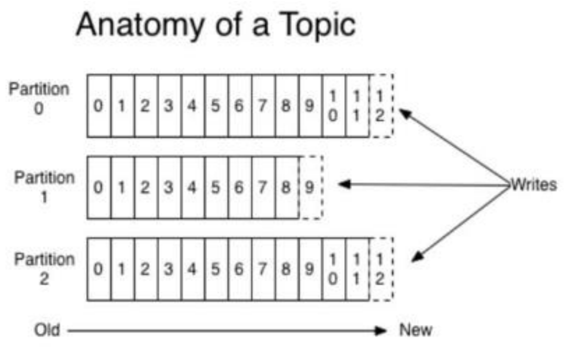


Fig. 11. Anatomy of a Topic [7]

The Kafka cluster stores streams of records in categories called topics. Each record consists of a key, a value, and a timestamp[12]. A topic is a category or feed name to which records are published. A topic is broken down into partitions that can be scaled horizontally. Topics in Kafka are always multi-subscriber; that is, a topic can have zero, one, or many consumers that subscribe to the data written to it. In our architecture, the Kafka Producer produces continuous streams of data in the form of clickstream events and pushes them on to the Kafka broker. The streams are then forwarded to HDFS and Spark's stream processing engine.

C) APACHE KAFKA – BATCH + SPEED

In the previous architecture, there was a mixture of two processing systems for the batch and speed layer respectively. The batch layer consisted of the Hadoop stack –HDFS for storage and MapReduce for compute. The speed layer consisted of the fast MapReduce version, i.e. Apache Storm. In our architecture, we will rely only on Apache Spark. Spark revolves around the concept of a resilient distributed dataset (RDD), which is a fault-tolerant collection of elements that can be operated on in parallel[8].An RDD is immutable data structure. There are two types of operations for an RDD, i.e. transformations and actions. A transformation basically creates a new dataset from an existing dataset. An action basically after running a computation on the dataset, returns a value to the driver program. Spark uses a DAG approach to optimize the operations. In Spark, the transformations are lazy and are not computed until an action operation occurs. Whenever an action takes place, all the transformations are executed. A directed acyclic graph is formed for the RDDs and are executed in order. If a dataset is created through a map transformation operation then it will only return the result of the reduce to the driver instead of a larger mapped dataset. By default, each transformed RDD may be recomputed each time you run an action on it. However, you may also persist an RDD in memory using the persist (or cache) method, in which case Spark will keep the elements around on the cluster for much faster access the next time you query it. We make use of the Kafka Consumer APIs to Integrate Kafka with Spark. There are two types of approaches to achieve the same, i.e. Receiver based Approach and Direct Approach. We will use the Direct Approach because of the following advantages it provides:1)Efficiency2)SimplifiedparallelismEx actly-onesemanticsAlso, Direct Kafka streams will make sure that there is a 1-1 mapping between the Kafka partition and the Spark partition. We can make use of Spark's Data Frame API to read data from HDFS in a structured format using Spark SQL. Building the Speed Layer with Spark: Basically, we fork the data coming from Kafka into a HDFS and into stream processing engine of Kafka. We make use of Spark's DStream API to deal with the streaming data. Discretized Stream or DStream is the basic abstraction provided by Spark Streaming. It represents a continuous stream of data, either the input data stream received from source, or the processed data stream generated by transforming the input stream. Internally, a DStream is represented by a continuous series of RDDs, which is Spark's abstraction of an immutable, distributed dataset. Each RDD in a DStream contains data from a certain interval. Any operation applied on a DStream translates to operations on the underlying RDDs. The DStream operations hide most of these details and provide the developer with a higher-level API for convenience.

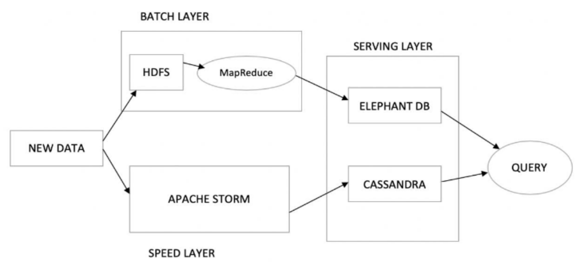
D) APACHE KAFKA – SERVING LAYER

Apache Cassandra is a NoSQL distributed management system designed for large amounts of data and provides high availability with no single point of failure. It serves our use case as it can take heavy write loads and be gracefully used for analytics. We use a Spark Cassandra Connector Library that provides excellent integration between the two systems. As per the CAP theorem, it is only possible to have either of consistency or availability in a distributed system. The consistency parameter in Cassandra is tunable and configurable with a trade-off with availability. In our use case, there is a need for high availability and hence, consistency can be compromised. Furthermore, the data modeling in Cassandra revolves around the queries. It is possible to build tailored tables and materialized views with a read-write separation for better performance. A Cassandra cluster is configurable and for our use case I went with a simple strategy with a default replication factor of 3. Additionally, Cassandra's Query Language (CQL) integrates well with Apache Zeppelin for querying and analytics.

E) APACHE ZEPPELIN

Apache Zeppelin is a web-based notebook which brings data exploration, visualization, sharing and collaboration features to Spark. The reason to choose Zeppelin is because of its support for Spark APIs and CQL .

VIII. COMPARATIVE ANALYSIS



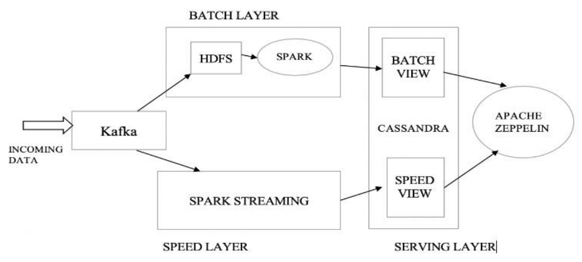


Fig. 12. COMPARISON BETWEEN ARCHITECTURES

MAPREDUCE VS SPARK

As demonstrated in the table below, Spark gives better performance compared to MapReduce

DATASET SIZE (size in MB)	MAPREDUCE (execution time in seconds)	APACHE SPARK (execution time in seconds)
80	46	20
263	79	41
550	120	73
800	183	107
1230	288	157

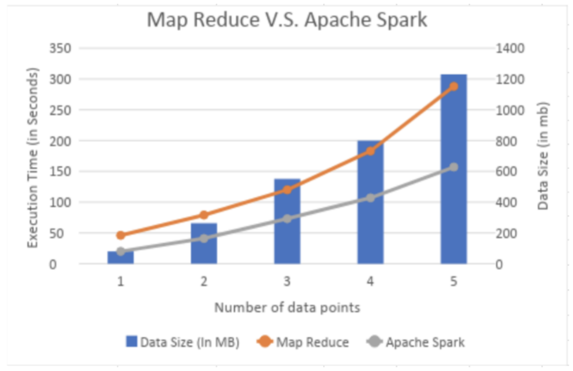


Fig. 13. MapReduce vs Apache Spark

SPARK VS STORM

When the Lambda Architecture was initially proposed, it consisted of Apache Storm as the engine for the speed layer. Apache Storm follows a native streaming approach for the creation of real-time views. Spark has grown exponentially during these years and has been a top-level Apache Project. Spark has been a preferred solution due to rich APIs for SQL (Spark SQL) and Machine Learning (MLlib). Spark streaming is more of a micro-batch processing engine. The below diagram demonstrates the meaning of micro-batching as it is an overlap of batch and stream and can be thought of as a fast-batch processing.

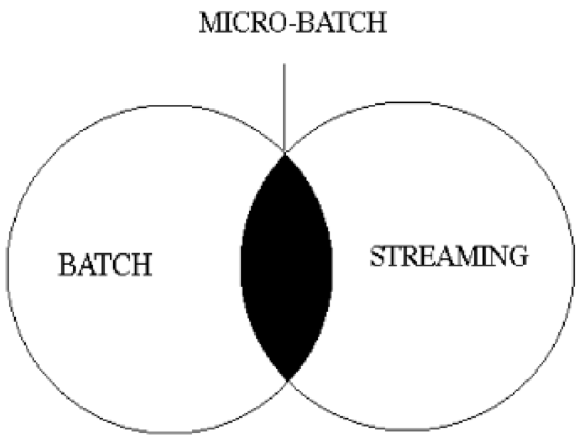


Fig. 14. Batch vs Streaming vs Micro-Batch

The choice between Storm and Spark Streaming is dependent on a lot of factors. Both the technologies have built-in support for fault tolerance and high availability. The term real-time data processing is a relative term and dependent on the use case. For our use case, initially, Storm performs slightly better in comparison to Spark on the basis of low-latency. As time progresses, Spark Streaming matches the performance of Storm. There is a trade-off between latency and ease-of-development that impacts the design decision in the proposed architecture. Unlike Spark, Storm does not support using the same application code for both batch and stream processing. The traditional architecture has separate processing engines across the parallel layers. There is a considerable overhead in software installation and maintenance of MapReduce and Storm. Even though the application logic remains almost same, writing code for two different systems was difficult for my use case. The group of APIs supported by Storm and MapReduce are very different to each other. This issue can easily scale up on an industry-level. This is where Spark's unified API for batch and stream processing helped my case. Spark has a unified API that can be reused in both the layers. Spark Streaming is inherently a micro-batch, therefore, for each RDD operation streamlined the application logic. The proposed architecture performs computation and manipulation in SQL with the help of SparkSQL. This helps the vast community of developers and analysts that prefer SQL for data processing. Additionally, the interoperability between Spark and Cassandra is smooth due to the Spark-Cassandra connector. This helped us push the processed records into Cassandra tables with ease. We carried out an analysis for different data sizes and worked it out with Apache Storm and Spark Streaming. The demonstration can be seen below in the table followed by a figure.

DATASET SIZE (size in MB)	APACHE STORM (execution time in seconds)	SPARK STREAMING (execution time in seconds)
600	0.41	0.62
800	0.53	0.67
1200	0.58	0.71
1350	1.07	1.09
1570	1.12	1.15

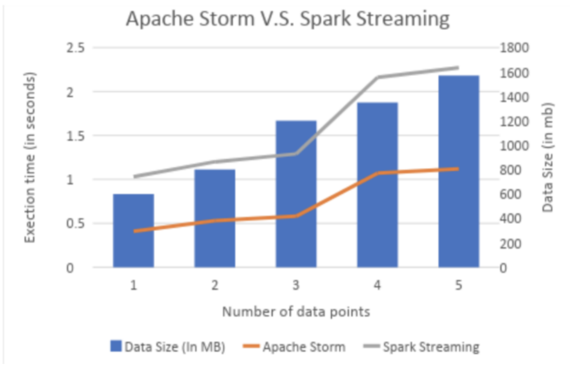


Fig. 15. Storm vs Spark Streaming

KAFKA VS OTHER QUEUEING SYSTEMS

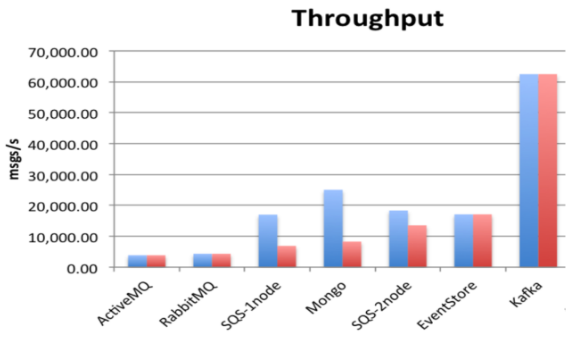


Fig. 16. Performance Comparison Between Queueing Systems [21]

The above figure is a demonstration for the comparison between different pub-sub/queueing systems. The blue value represents the throughput of the sending messages i.e. Sender. The red value represents the throughput of receiving the messages i.e. Receiver. The LA is meant to handle data at a huge scale. Especially for big data with very high velocity, a resilient queueing system such as Kafka is necessary. Even though other pub/sub systems such as ActiveMQ and RabbitMQ are used widely, they do not match the results of Kafka. The major reason to go with Kafka for our use case is due to its ability to rewind records with the help of an offset handled by a consumer. This offloads a lot of burden from the queueing system and results in loose coupling. Additionally, Kafka provides a very good persistence mechanism and a configurable retention period that fits in our use case very well. I have set the retention period to seven days after which I push the data to HDFS. If re-computation is required, we can always look up the data from HDFS.

SERVING LAYER SIMPLIFIED

In the traditional LA, the serving layer comprised of both the ElephantDB and Apache Cassandra. The views computed in the batch layer are sent to ElephantDB. The views calculated in the speed layer are sent to Apache Cassandra. The proposed architecture comprises of a single database to hold the views from both the batch and the speed layer. Due to growth and acceptance of Apache Cassandra, building software systems using Cassandra as a data store has become very easy. Unlike the traditional architecture, the proposed architecture has a Spark Processing engine on top of the serving layer datastore. Hence, a distributed, heavy-write database like Cassandra smoothly integrates with our use case. This alleviates the burden of maintaining separate databases and therefore reduces storage cost.

SUMMARIZED COMPARISON BETWEEN TRADITIONAL & PROPOSED ARCHITECTURE

	Traditional Lambda Architecture	Proposed Lambda Architecture
Batch Layer	Storage – HDFS Compute - MapReduce	Storage – Kafka (HDFS only if necessary) Compute – Apache Spark
Speed Layer	Apache Storm	Spark Streaming
Serving Layer	ElephantDB, Apache Cassandra	Apache Cassandra
Code Reusability	Low	High
Complexity	High	Low
Maintenance	High	Low
Learning Curve	High	Low
Cost	High	Low

IX. KEY SIGHTS

1) The trouble and overhead to duplicate incoming data can be eliminated with the help of Kafka. Kafka can store messages over a period of time that can be consumed by multiple consumers. Each consumer maintains an offset to denote a position in the commit log. Thus, the batch and speed layers of the LA can act as consumers to the published records on the topics in Kafka. This reduces the duplication complexity and simplifies the architecture.
2) Spark's rich APIs for batch and streaming make it a tailor-made solution for the LA. The underlying element for Spark streaming API (D Streams) is a collection of RDDs. Hence, there is huge scope for code reuse. Also, the business logic is simpler to maintain and debug. Additionally, for the batch layer, Spark is a better option due to the speed it achieves because of in-memory processing.
3) Kafka is a beast that can store messages for as long as the use case demands (7 days by default, but that is configurable). We have used HDFS for reliability in case of human or machine faults. In fact, we can totally eliminate HDFS from the batch layer and store the records as they are in the Kafka topics. Whenever re-computation is required, the commit log inside Kafka can be replayed.
4) Kafka's commit log is very useful for event sourcing as well. As the commit log is an immutable, append-only data store, a history of user events can be tracked for analytics. It finds it application in online retail and recommendationengines.
5) Cassandra is a write heavy database that can build tables as per new use cases. Kafka commit log can be replayed and new views can be materialized using Cassandra.

X. CONCLUSION& FUTURE WORK

Due to a tremendous evolution of big data, there is a big challenge to process and analyze the large amounts of data. Traditional systems such as relational databases and batch processing systems are not able to keep up with the big data trends. Even though Hadoop frameworks instill a great promise and reduce complexities in distributed systems, they fail to satisfy the real-time processing capabilities. The primary goal of the project is to demonstrate and synthesize the advancements in the field of real-time data processing with Lambda architecture. The architecture is a robust combination of batch and stream processing techniques. Each layer in the architecture has a defined role and is decoupled from the other layer. The LA is overall a highly distributed, scalable, fault-tolerant, low-latency platform. All the data entering the system is sent to both the batch and the speed layer for further processing. The purpose of the batch layer is management of the master dataset and a pre-compute of the batch views. The high latency disadvantage of the batch layer is compensated by a speed layer. The serving layer accumulates the batch and real-time views. The beauty of the architecture is to apply different technologies across the three layers. As the big data tools are improving day-by-day, the LA provides new opportunities and possibilities. It is the responsibility of the LA to translate the incoming raw data into something meaningful. There is a massive demand for real-time data processing in the industry, which is addressed by the LA. The system is a good balance of speed and reliability. Although the LA has many pros, there are a few observed cons as well. There area lot of moving parts resulting into coding overhead due to the involvement of comprehensive processing in the disparate batch and speed layers. In certain scenarios, it is not beneficial to perform re-processing for every batch cycle. Additionally, it can be difficult to merge the views in the serving layer. Therefore, we have introduced a LA using Apache Spark as a common framework for both batch and speed layer. This simplifies the architecture to a great extent and is maintainable as well. Furthermore, a detailed comparison has been made between the two architectures depending on the design decision and trade-offs. The future work will be to understand the different technologies that can be used as an alternative to the proposed architecture. Apache Samza is a stream processing engine that integrates very well with Kafka. It is built by the same researchers that were responsible for the development of Kafka. It would be interesting to try different use cases with the architecture and analyze how they perform.

ANNEXURE: FIGURES

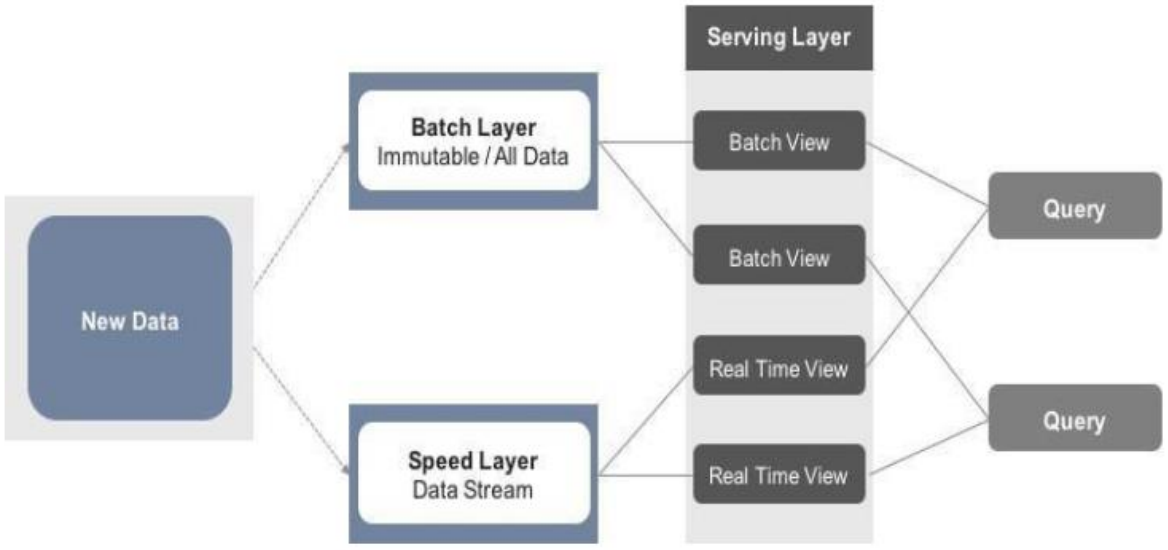


Fig. 1. Lambda Architecture [2]

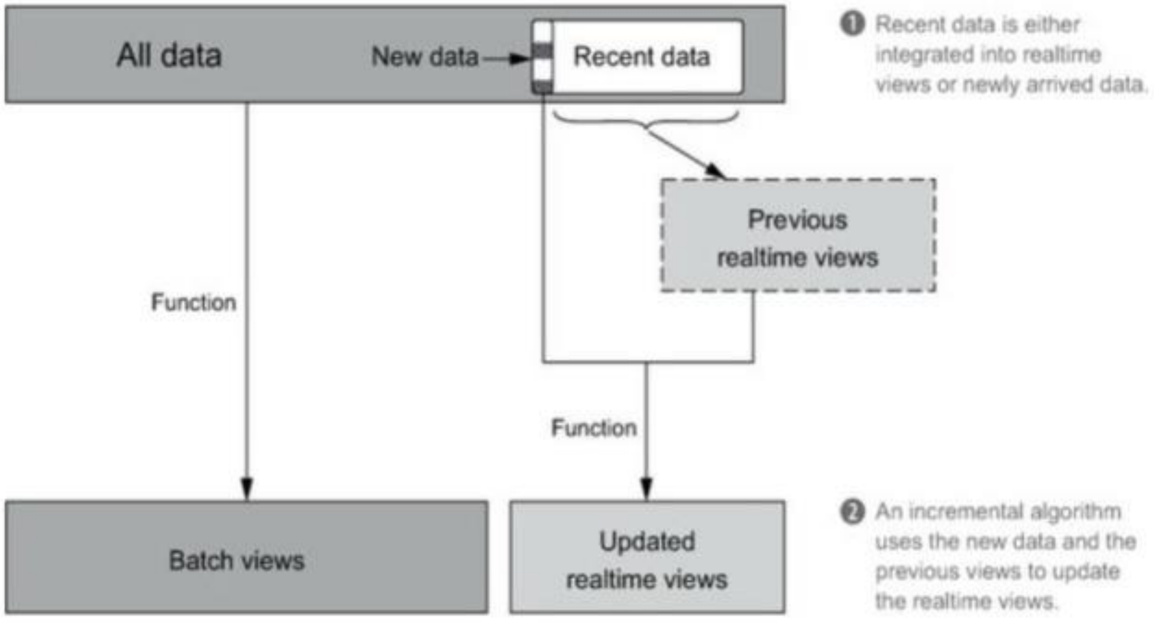

All data
New data
Recent data
1 Recent data is either integrated into realtime views or newly arrived data.
Previous realtime views
Function
Function
Batch views
Updated realtime views
2 An incremental algorithm uses the new data and the previous views to update the realtime views.

Fig. 2. Incremental Computation Strategy [3]

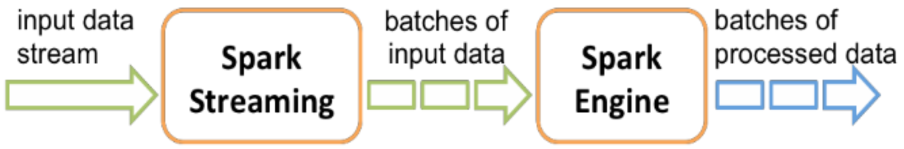


Fig. 3. Spark Streaming [10]

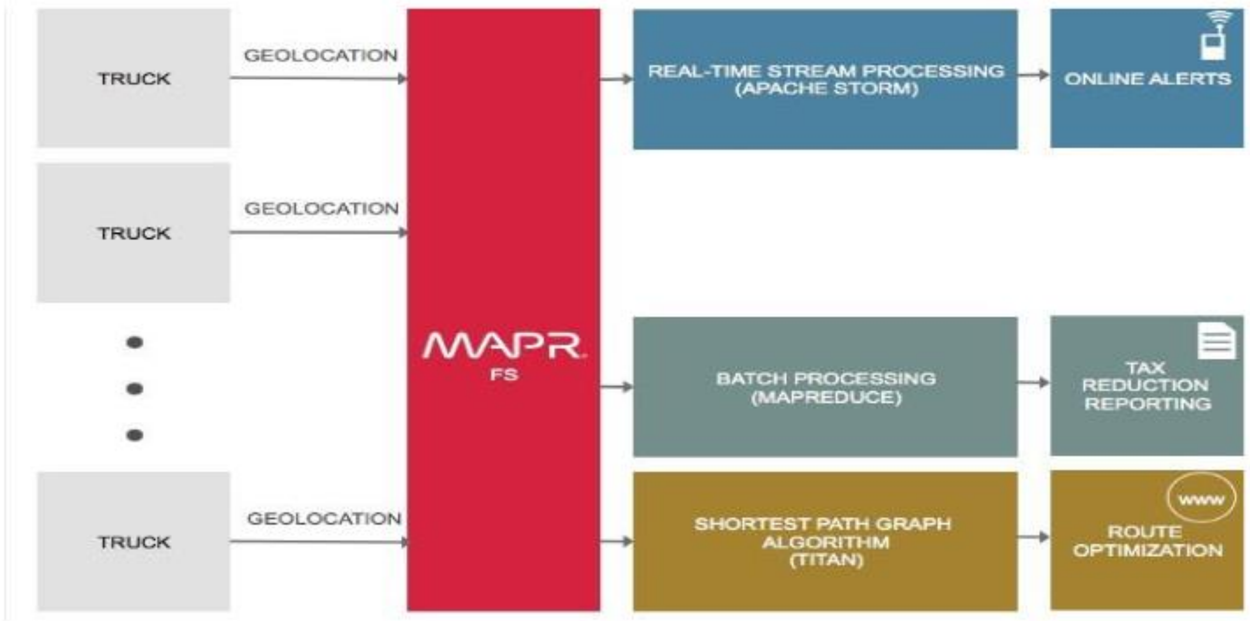


Fig. 4. Lambda Architecture at MapR [14]

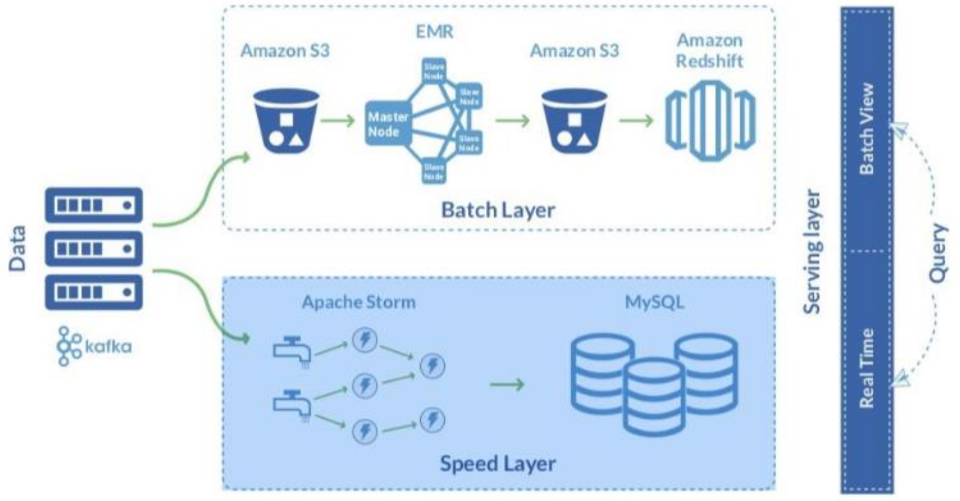


Fig. 5. Lambda Architecture at Talentica[15]

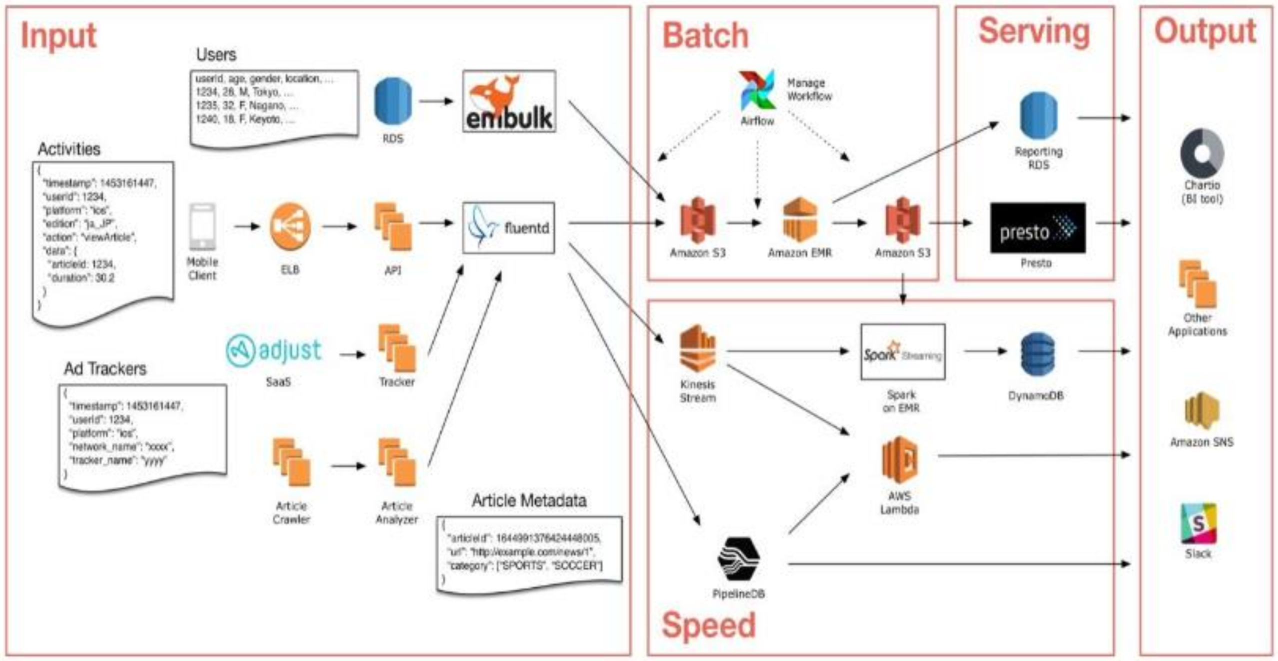


Fig. 5. Lambda Architecture at Talentica [16]

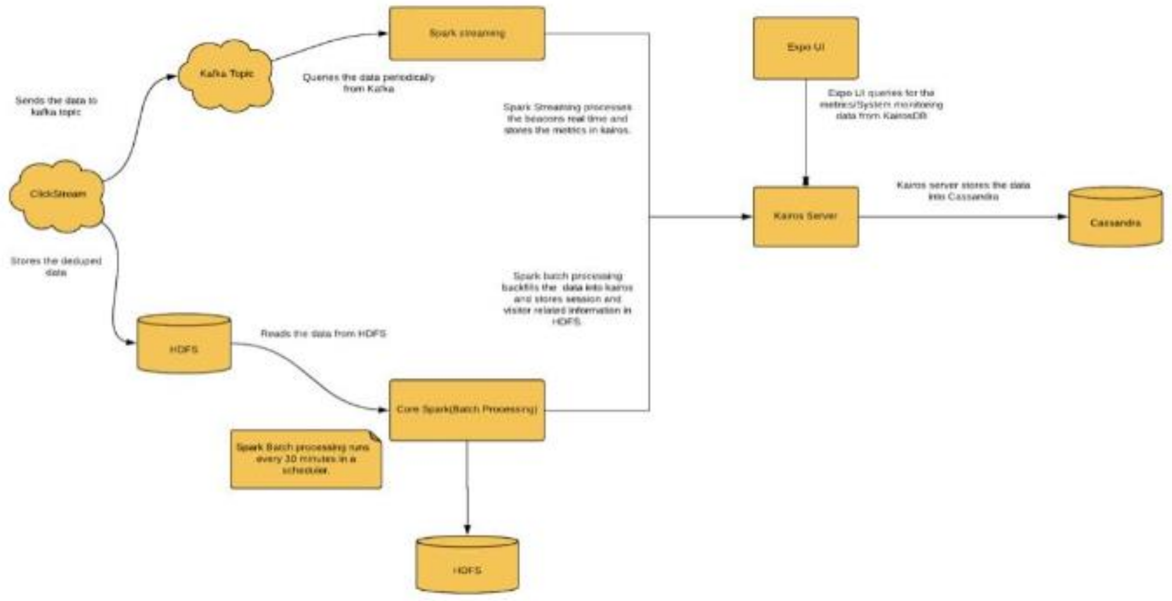

Spark streaming
Kafka Topic
Queries the data periodically from Kafka
Sends the data to kafka topic
ClickStream
Spark Streaming processes the beacons real time and stores the metrics in kairos.
Expo UI
Expo UI queries for the metrics/System monitoring data from KairosDB
Kairos Server
Kairos server stores the data into Cassandra
Cassandra
Stores the deduped data
HDFS
Reads the data from HDFS
Spark batch processing backfills the data into kairos and stores session and visitor related information in HDFS.
Core Spark(Batch Processing)
Spark Batch processing runs every 30 minutes in a scheduler.
HDFS

Fig. 6. Lambda Architecture at Walmart Labs [17]

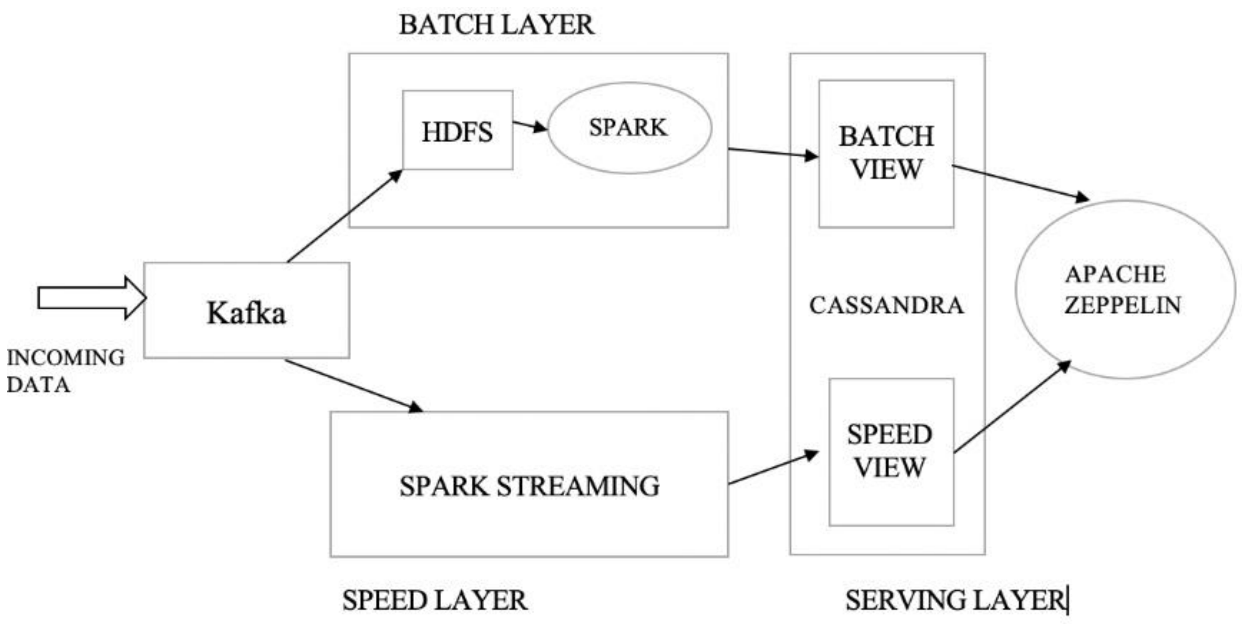


Fig. 7. Proposed LAMBDA Architecture [18]

REFERENCES

[1] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” Proc. 6th Symp. Oper. Syst. Des. Implement., pp. 137–149,2004.

[2] T. Chen, D. Dai, Y. Huang, and X. Zhou, “MapReduce On Stream Processing,” pp. 337–341.

[3] N. Marz and J. Warren. Big Data: Principles and best practices of scalable real time data systems. Manning Publications Co.,2015.

[4] J. Kreps, “Questioning the Lambda Architecture”, https://www.oreilly.com/ideas/questioning-the-lambda-architecture

[5] I. Ganelin, E. Orhian, K. Sasaki, and B. York, Spark: Big Data Cluster Computing in Production.2016.

[6] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)(MSST '10). IEEE Computer Society, Washington, DC, USA, 1-10. DOI=http://dx.doi.org/10.1109/MSST.2010.5496972

[7] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a distributed messaging system for log processing. SIGMOD Workshop on Networking Meets Databases, 2011.

[8] W. Yang et al. “Big Data Real-Time Processing Based on Storm.” 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (2013): 1784-1787.

[9] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. Commun. ACM 59, 11 (October 2016), 56-65. DOI: https://doi.org/10.1145/2934664

[10] Apache Spark, spark.apache.org

[11] Apache Storm, www.storm.apache.org

[12] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. 44, 2 (April 2010), 35-40. DOI=http://dx.doi.org/10.1145/1773912.1773922

[13] Apache HBase, https://hbase.apache.org

[14] “Stream processing with MapR”, https://mapr.com/resources/stream-processing-mapr/

[15] https://www.talentica.com/assets/white-papers/big-data-using-lambda-architecture

[16] https://aws.amazon.com/blogs/big-data/how-smartnews-built-a-lambda-architecture-on-aws-to-analyze-customer-behavior-and-recommend-content/

[17] S. Kasireddy, “How we built a data pipeline with Lambda Architecture using Spark/Spark Streaming”,https://medium.com/walmartlabs/how-we-built-a-data-pipeline-with-lambda-architecture-using-spark-spark-streaming-9d3b4b4555d3

[18] “Lambda Architecture for Batch and Stream Processing”,https://d0.awsstatic.com/whitepapers/lambda-architecure-on-for-batch-aws.pdf

[19] “A prototype of Lambda architecture for financial risk”,https://www.linkedin.com/pulse/prototype-lambda-architecture-build-financial-risk-mich/

[20] “Applying the Lambda Architecture with Spark, Kafka, andCassandra”,https://github.com/aalkilani/spark-kafka-cassandra-applying-lambda-architecture

[21] Adam Warski, “Evaluating persistent, replicated message queues”http://www.warski.org/blog/2014/07/evaluating-persistent-replicated-message-queues/

AUTHORS:

*First Author – Dr. Yuvraj Kumar, Ph.D. in Artificial Intelligence, Faculty of Quantum Computing and Artificial Intelligence, Zukovsky State University, Russia