

# IMPLEMENTING A LAMBDA ARCHITECTURE TO PERFORM REAL-TIME UPDATES

by

PRAMOD KUMAR GUDIPATI

B.E., OSMANIA UNIVERSITY (OU), INDIA, 2012

---

A REPORT

submitted in partial fulfillment of the  
requirements of the degree

MASTER OF SCIENCE

Department of Computing and Information Science  
College of Engineering

KANSAS STATE UNIVERSITY

Manhattan, Kansas

2016

Approved by:

Major Professor  
Dr. William Hsu

# COPYRIGHT

Pramod Kumar Gudipati

2016

# Abstract

The Lambda Architecture is the new paradigm for big data, that helps in data processing with a balance on throughput, latency and fault-tolerance. There exists no single tool that provides a complete solution in terms of better accuracy, low latency and high throughput. This initiated the idea to use a set of tools and techniques to build a complete big data system. The Lambda Architecture defines a set of layers to fit in a set of tools and techniques rightly for building a complete big data system: Speed Layer, Serving Layer, Batch Layer. Each layer satisfies a set of properties and builds upon the functionality provided by the layers beneath it.

The Batch layer is the place where the master dataset is stored, which is an immutable and append-only set of raw data. Also, batch layer pre-computes results using a distributed processing system like Hadoop, Apache Spark that can handle large quantities of data. The Speed Layer captures new data coming in real time and processes it. The Serving Layer contains a parallel processing query engine, which takes results from both Batch and Speed layers and responds to queries in real time with low latency.

Stack Overflow is a Question & Answer forum with a huge user community, millions of posts with a rapid growth over the years. This project demonstrates The Lambda Architecture by constructing a data pipeline, to add a new “Recommended Questions” section in Stack Overflow user profile and update the questions suggested in real time. Also, various statistics such as trending tags, user performance numbers such as UpVotes, DownVotes are shown in user dashboard by querying through batch processing layer.

# Table of Contents

<b>List of Figures .....</b>	<b>V</b>
<b>Acknowledgements.....</b>	<b>VI</b>
<b>Chapter 1 - Introduction .....</b>	<b>1</b>
<b>Chapter 2 - Background and Related Work .....</b>	<b>2</b>
<b>Chapter 3 - Architecture .....</b>	<b>5</b>
<b>3.1 Batch Layer .....</b>	<b>5</b>
3.1.1 HDFS.....	5
3.1.2 Apache Spark.....	6
<b>3.2 Speed Layer.....</b>	<b>6</b>
<b>3.3 Serving Layer .....</b>	<b>8</b>
<b>3.4 Flask .....</b>	<b>9</b>
<b>Chapter 4 - Implementation.....</b>	<b>10</b>
<b>4.1 DataSet .....</b>	<b>10</b>
<b>4.2 Batch Layer .....</b>	<b>11</b>
<b>4.3 Data Ingestion .....</b>	<b>12</b>
<b>4.4 Speed Layer.....</b>	<b>13</b>
<b>4.5 Cassandra.....</b>	<b>13</b>
<b>Chapter 5 - Results.....</b>	<b>15</b>
<b>Chapter 6 - Conclusion &amp; Future Work.....</b>	<b>18</b>
<b>6.1 Conclusion .....</b>	<b>18</b>
<b>6.2 Future Work .....</b>	<b>18</b>
<b>Bibliography .....</b>	<b>19</b>

# List of Figures

Figure 1: Lambda Architecture .....	2
Figure 2: Lambda Architecture Diagram .....	4
Figure 3: HDFS Architecture. Adapted from Apache Hadoop .....	5
Figure 4: Apache Spark Ecosystem. Adapted from Apache Spark .....	6
Figure 5: Spark Streaming Architecture. Adapted from DataBricks .....	7
Figure 6: Kafka Architecture.....	8
Figure 7: Data Pipeline.....	10
Figure 8: Raw Data Format.....	11
Figure 9: Apache Spark setup on AWS Cluster .....	12
Figure 10: Kafka, Flask Cluster on AWS.....	13
Figure 11: Cassandra cluster on AWS.....	14
Figure 12: Application Home Page .....	15
Figure 13: User Dashboard Page.....	16
Figure 14: Top Tags .....	17
Figure 15: Trending Tags.....	17

# Acknowledgements

I would like to express my gratitude to my advisor, Dr. William Hsu, for his constant support and encouragement in completing this project. I would also like to thank Dr. Mitch Neilsen and Dr. Torben Amtoft for serving on my M.S. committee. Finally, I thank my family and friends for their love and support.

# Chapter 1 - Introduction

In the last decade, the amount of data being created has increased enormously. In the internet, on an average for every second 7000 tweets are made on twitter, 5000 statuses are updated on Facebook, 746 Instagram photos uploaded, more than 30,000 gigabytes of data is generated and the rate of data creation is only boosting up. There is a vast diversity in the data being generated like photos, blog posts, videos, tweets and servers log a message for each event occurring which would help in loading history back if needed. As a result, the internet is almost incomprehensibly large. Traditional data base systems like relational databases fail to handle these large amounts of data. This resulted in the raise of new technologies grouped under the term NoSQL. These systems can scale up to large amounts of data and can handle complexity issues easily, as the databases and computation systems used for big data are aware of their distributed nature.

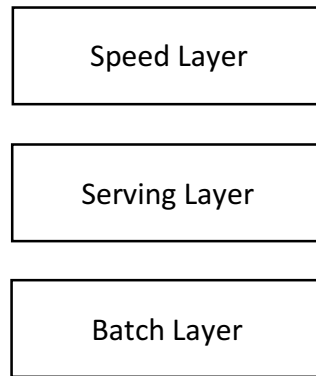
The most important technique with NoSQL systems is making data immutable. At any point data is not mutated, instead stored as it is in the system, making it possible to revert back if you make a mistake and write a bad data. This is a much stronger human fault-tolerant system than a traditional systems operated based on data mutation. Traditional systems can also store immutable data but they can't scale up with larger inputs of data coming in for storage.

Last decade has seen huge amount of innovation in scalable data systems, with the invention of large-scale computation systems like Apache Hadoop, Apache Spark and databases such as Cassandra, HBase and Riak. For instance, Hadoop can process large amounts of data with parallelized computations but with higher latency. Databases like Cassandra offer scalability through distributed nature by offering a confined data model compared to traditional database systems. It is very complex to fit your application data model into these systems. Each tool has its own trade-offs, but when these tools used in conjunction, one can develop scalable systems with low latency, human fault-tolerance and a minimum of complexity. The Lambda Architecture provides a general purpose approach that takes entire dataset as input and answers your questions with low latency. The set of technologies used may change based on the requirements of the application, but the lambda architecture defines a consistent approach in wiring those technologies together to build a system that meets your requirements.

There is no single tool that builds a complete data system with low latency, high throughput and human fault-tolerant. Instead, a set of tools and techniques should be used in conjunction to build a complete data system.

# Chapter 2 - Background and Related Work

The Lambda architecture defines the conjunction of different tools as a series of layers and each layer satisfies a set of properties and built on top of the functionality provided by the layer beneath it.



*Figure 1: Lambda Architecture*

The entire dataset is passed as input to the system on which you run a query to retrieve the information needed, which can be generalized as

$$\text{query} = \text{function}(\text{all data})$$

If the input data is of size 1 petabyte, for each input query you want to answer, you can run the function and get the results. But processing huge data for each query not only takes a large amount of resources but also unreasonably expensive. An alternative approach would be to pre-compute the query function; the result would be called as a batch view. Instead of computing the results on the fly, you could read batch view to get results.

$$\text{batch view} = \text{function}(\text{all data})$$

$$\text{query} = \text{function}(\text{batch view})$$

The batch view makes it possible to get results quickly without having to process entire data set each time. Also, do remember that creating batch view is a high latency operation as it processes entire data set, during which a lot of new data gets created which is not considered while creating the batch view and results are out of data for few hours.

**Batch Layer** part of the Lambda architecture is responsible for creating the batch view. The



master data set is stored in the batch layer and creates pre-compute views on the master data set. The two tasks of the batch layer are: to store an immutable growing master dataset and compute arbitrary functions on the data set to create batch views. This type of computations is done by batch processing systems like Hadoop, Apache Spark.

**The Serving Layer** loads the batch views that are produced by batch layer as a result of its functions. The Serving Layer is a specialized distributed database that loads batch views and provides random access to the batch view through queries. The database should be able to update batch views coming from the batch layer periodically and allow random reads. Random writes are not important for the serving layer databases, reducing the most complex part of the database. Thus, these databases are simple making them easy to operate, configure and robust. ElephantDB is an example of a database that can be used in the serving layer.

**The Speed Layer** is responsible for handling low latency updates, which occur due to high latency batch view creation in the batch layer. The serving layer updates when the batch layer completes computing a batch view, which means it misses processing the data that arrived during batch view creation. The Speed layer processes this missing data and ensures that new data is represented in query functions as quickly as needed based on the applications need.

The Speed layer is similar to the batch layer in that it computes views based on data it receives. The only difference being the speed layer just considers recent input data for processing, whereas batch layer takes all the data. As it receives the data, it updates the real-time views and does perform an incremental computation. We can generalize the speed layer computation into:

$$\text{realtimeview} = \text{function}(\text{realtimeview}, \text{new data})$$

The Lambda Architecture in total can be generalized into following:

$$\text{batch view} = \text{function}(\text{all data})$$

$$\text{realtimeview} = \text{function}(\text{realtimeview}, \text{new data})$$

$$\text{query} = \text{function}(\text{batchview}, \text{realtimeview})$$

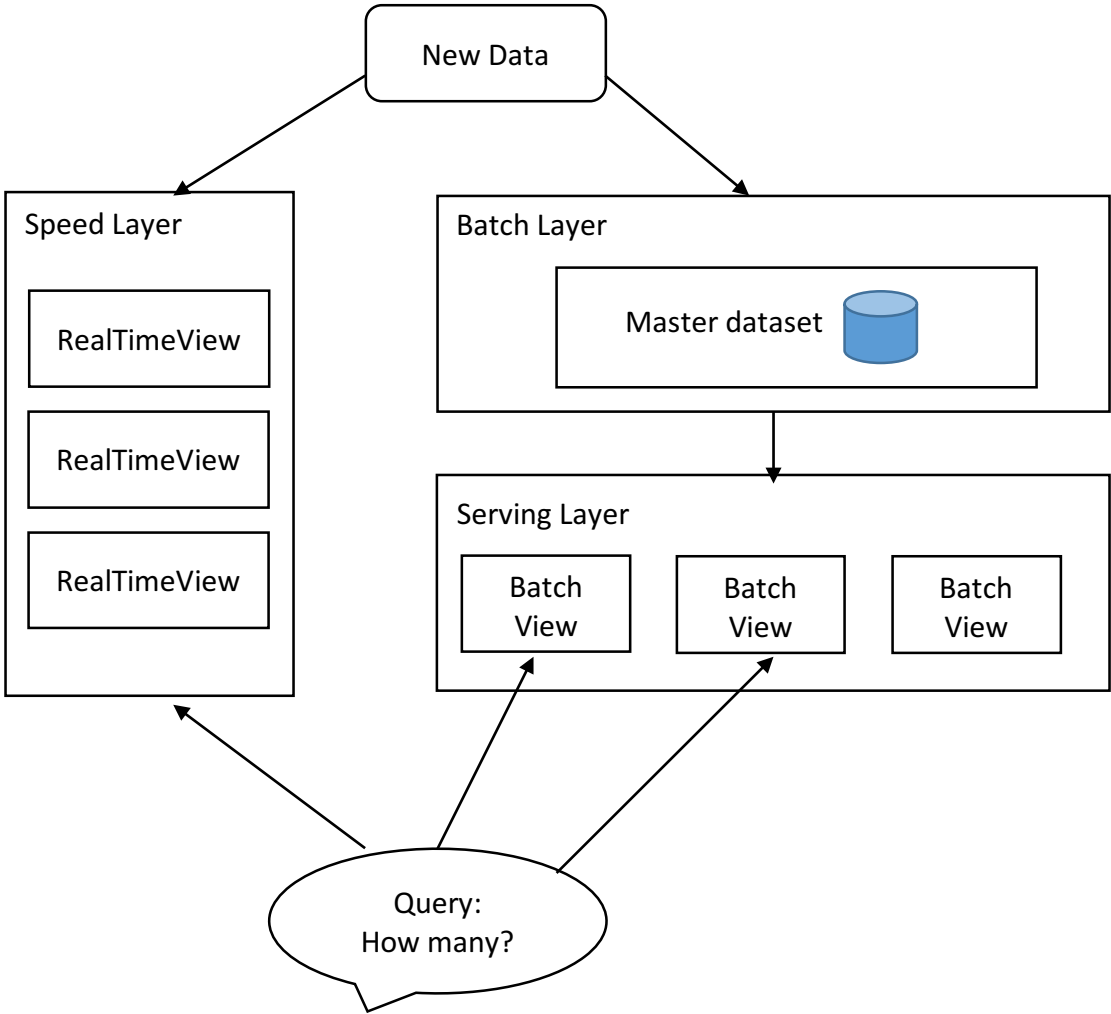


Figure 2: Lambda Architecture Diagram

# Chapter 3 - Architecture

To implement the Lambda architecture, we need different tools at each layer to perform the tasks at each layer.

## 3.1 Batch Layer

The Hadoop Distributed File System (HDFS) and Apache Spark are the technologies/tools which I have used for my batch layer portion.

### 3.1.1 HDFS

It is a distributed, scalable and portable file system. It can store large files typically from gigabytes to terabytes. It achieves reliability by replicating data across multiple nodes. HDFS has a master/slave architecture. An HDFS cluster consists of a single NameNode, which acts as a master server that manages the file system namespace and regulates the access to files by clients. A set of DataNodes is present in the cluster, that store blocks of data achieved by splitting input files into blocks of data. DataNodes are responsible for serving read and write requests from the file system's clients. The NameNode is responsible for handling operations like data blocks to DataNodes mapping, opening, closing, renaming files and directories.

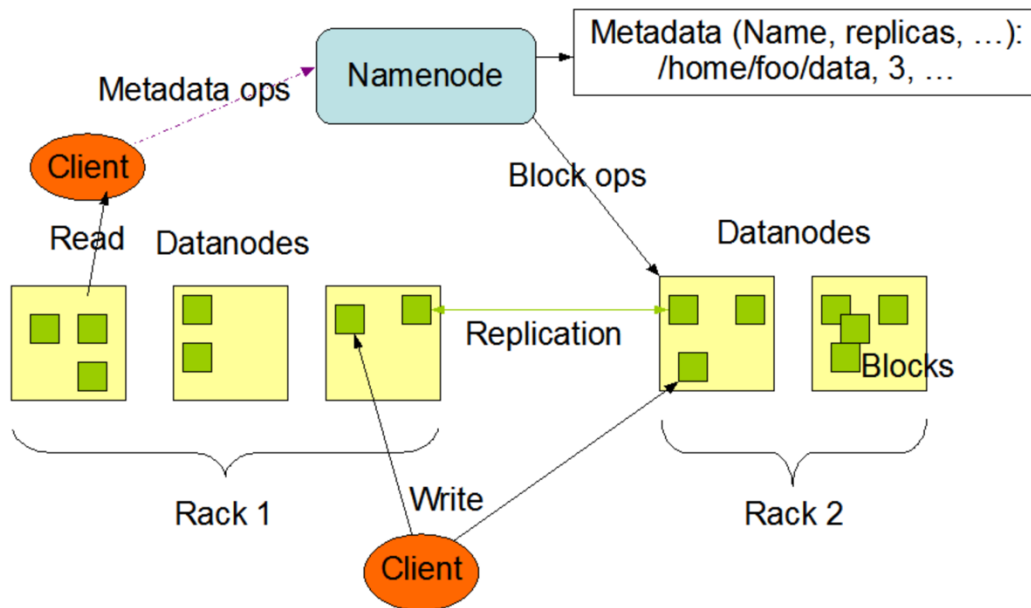


Figure 3: HDFS Architecture. Adapted from Apache Hadoop

HDFS stores each file as a sequence of blocks of the same size, except the last block size. It provides fault tolerance by replicating blocks of data across different DataNodes. The block size and replication factor are configurable. The NameNode handles replication of data blocks and

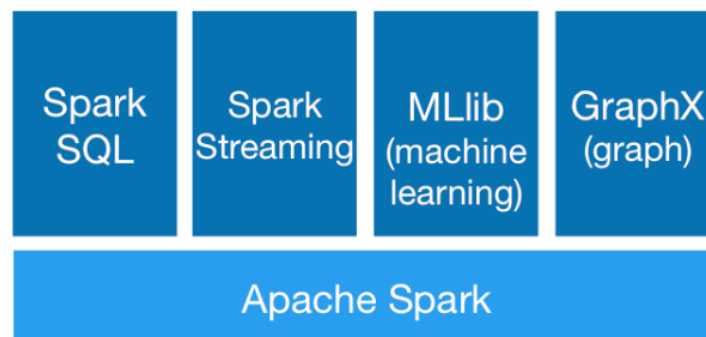
periodically receives a heartbeat from DataNode as an acknowledgment that DataNode is alive and working.

### 3.1.2 Apache Spark

It is an open source cluster computing framework, that provides a high-level API's built to perform various operations on data structure named Resilient Distributed DataSet(RDD). RDD is a read only multi-set of data items distributed over a cluster of machines that is maintained in a fault-tolerant way. It also supports a rich set of tools like Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing. These API's are available in Java, Scala, Python, and R.

Apache Spark has an advanced DAG execute engine that supports cyclic data flow and in-memory computing. Spark is very optimized for iterative operations on data set because the data set is cached in memory instead of accessing it from disk every time.

Transformations and Actions are the 2 types of operations performed on RDD's. Spark constructs a Directed Acyclic graph before performing the actual operation itself, performs all the actions at once after reaching the statement that performs save or show the result. This kind of evaluation is named as Lazy Evaluation, which optimized the spark performance.



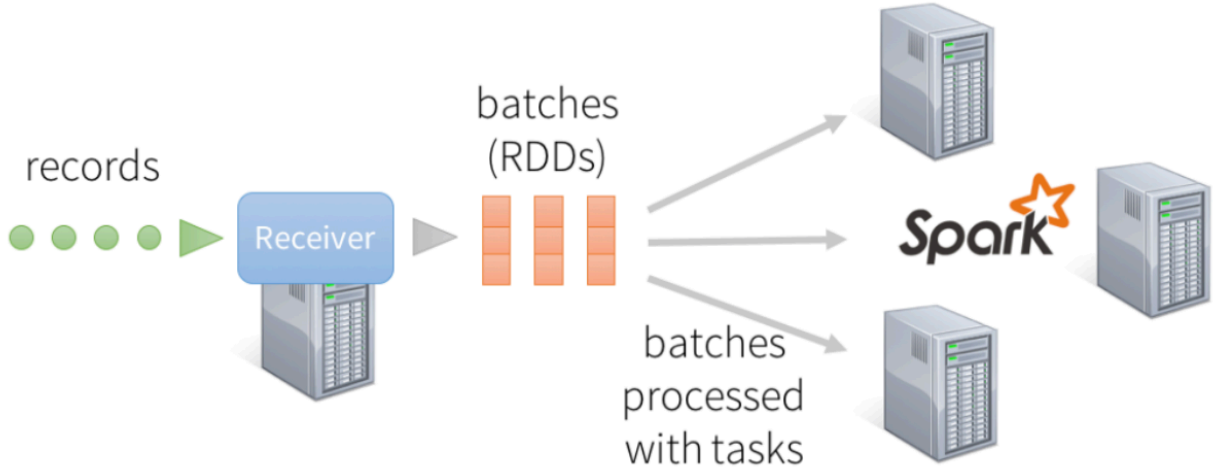
*Figure 4: Apache Spark Ecosystem. Adapted from Apache Spark*

Spark SQL is a tool built on top of Apache Spark, which is used for processing structured dataset. I have mainly used Spark SQL for batch processing. To use Spark SQL, we create temporary tables on top of input data and write SQL queries that run on temporary tables to retrieve the results in a distributed fashion.

### 3.2 Speed Layer

I have used Spark Streaming in the Speed layer for processing real-time data. Spark Streaming is an extension of core Spark API that enables scalable, high throughput processing capability on streaming data. Spark Streaming divides data into batches, which are processed further to

generate a stream of results in batches. It provides a high-level abstraction called Discretized Stream or DStream, which represents a continuous stream of data. DStream is internally represented as a sequence of RDDs. Data can be ingested to Spark Streaming from a variety of sources like Kafka, Flume, Twitter, Kinesis.

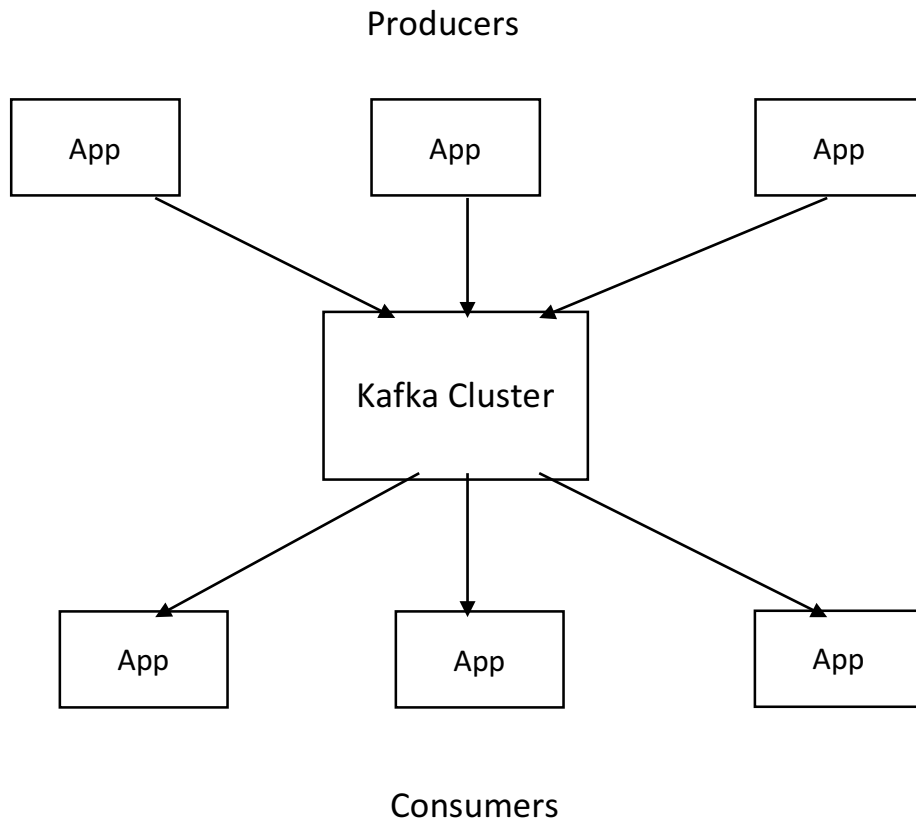


records processed in batches with short tasks  
each batch is a RDD (partitioned dataset)

*Figure 5: Spark Streaming Architecture. Adapted from DataBricks*

**Kafka** is a distributed streaming platform that helps in building real-time streaming data pipelines in a fault-tolerant way. Kafka architecture is a combination of Message Queue and Publish-Subscribe models and it is highly scalable. Kafka is run as a cluster on one or more of the servers.

Kafka clusters store streams of records categorized as topics. Kafka contains producers and consumers. Producers as the name suggests, are the ones who create streams of records and write to topics. Consumers read a stream of records from topics and processes the records. In our case, Spark Streaming acts as a consumer and reads records.



*Figure 6: Kafka Architecture*

### 3.3 Serving Layer

This project uses Cassandra as a database in the serving layer to service information requested from the batch view and real-time views. Cassandra is an open source distributed database management system that can handle large amounts of data providing high availability with no single point of failure. It provides robust support for clusters with asynchronous masterless replication. Cassandra follows a decentralized approach i.e. every node in the cluster has the same role. Data is distributed across all the nodes in the cluster since there is no master any node can service any request.

Cassandra supports data replication and replication strategies are configurable. Data is automatically replicated across nodes for fault-tolerance. New nodes can be added to existing clusters without any interruption, making it scalable to handle more read and write operations with no downtime. Failed nodes can be replaced with no downtime. Cassandra provides tunable and eventual consistency with a latency of hundreds of milliseconds. Cassandra Query Language(CQL) is similar to traditional Structured query language(SQL), a simple interface to access Cassandra.

Cassandra's Data Model is a hybrid of key-value and column-oriented databases. Its data model is a partitioned row store, partitioned based on hashing of the first component of the primary key called as partition key, the remaining component of a primary key is used for clustering known as the clustering key. KeySpace in Cassandra resembles schema in Relational database and column family resembles a table in an RDBMS.

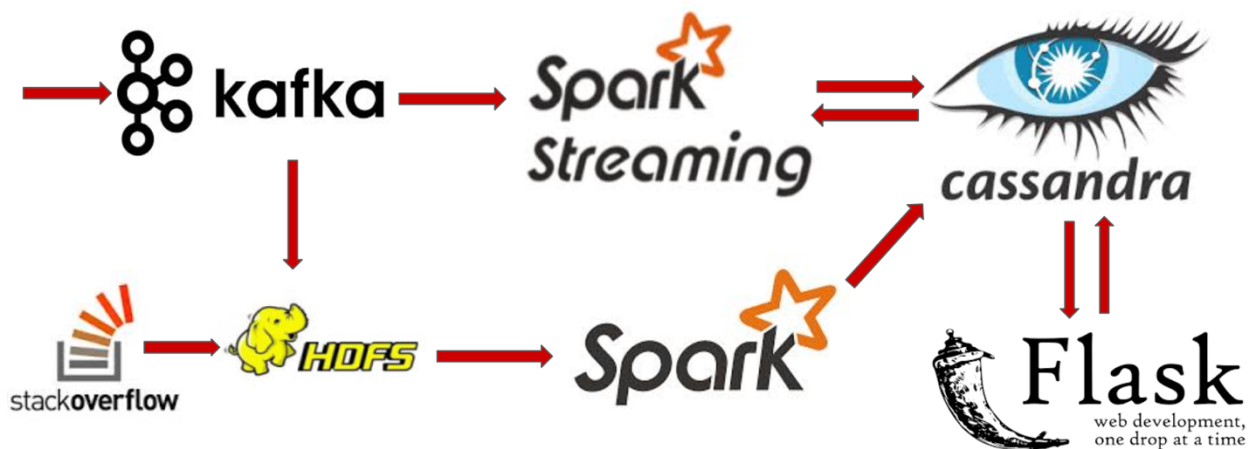
## 3.4 Flask

Flask is a micro web framework written in python. I have used flask for application development and hosted the application on Tornado web server. Tornado is a scalable, non-blocking web server and web application framework. With the use of non-blocking network I/O, Tornado scales to tens of thousands of open connections making it ideal for web sockets and applications that require long-lived connection to each user. Also, used HTML and Bootstrap to develop application pages.

# Chapter 4 - Implementation

The tools selected for each of those layers as explained in the earlier chapter are connected to create a data pipeline.

Figure 7 below depicts how the pipeline looks like:



*Figure 7: Data Pipeline*

## 4.1 DataSet

To implement the lambda architecture, we need a large dataset that requires batch and real-time processing. I opted for Stack Overflow dataset for my project. Stack Overflow releases data on a monthly basis, I took recent release of September month with data set size being around 200 GB. The complete application data is spread across different file types like users, posts, votes, badges. Data is in XML format with self-closing tags.

DataSet is downloaded from the Stack Overflow archive and stored in Hadoop File System(HDFS). Since data is in XML format with self-closing tags, existing XML libraries don't support processing XML with self-closing tags so I need to convert to CSV format to process the data.



### **Users.xml**

- `<row Id="1" Reputation="13260" CreationDate="2009-09-28T14:35:46.00" DisplayName="Anton Geraschenko" LastAccessDate="2015-10-15T16:21:58.72" WebsiteUrl="http://stacky.net" Location="Los Angeles, CA" AboutMe="&lt;p&gt;You can get in touch with me at geraschenko@gmail.com.&lt;/p&gt;&#xA;" Views="201" UpVotes="19" DownVotes="3" Age="33" AccountId="36500" />`

### **Posts.xml**

- `<row Id="1" PostTypeId="1" CreationDate="2013-06-25T03:05:29.92" Score="17" ViewCount="480" Body="&lt;p&gt;Where's the old meta.mathoverflow.net?&lt;/p&gt;&#xA;" OwnerDisplayName="user35287" LastEditorUserId="2000" LastEditDate="2013-09-28T12:00:12.54" LastActivityDate="2013-09-28T12:00:12.54" Title="Where's the old meta?" Tags="&lt;support&gt;&lt;faq&gt;" AnswerCount="1" CommentCount="5" />`

### **Votes.xml**

- `<row Id="251" PostId="60" VoteTypeId="5" UserId="35459" CreationDate="2013-06-25T00:00:00.00" />`

*Figure 8: Raw Data Format*

The above figure shows sample record from each of the users, posts, votes files and the highlighted attributes are the ones which I used for processing.

I would like to process following information from the dataset:

1. Identify top tags for each user,
2. Update user performance numbers like UpVotes, DownVotes in real time
3. Recommend Questions to users in real time using top tags,
4. Trending Tags in the application and update their counts in real time.

## 4.2 Batch Layer

I need to obtain above mentioned information from the dataset in the batch layer and update them in real time using the speed layer.

Since the data is in XML format, I have used Scala XML library and used Spark distributed nature to parse data into CSV format. Helper files are used to remove top 2 and last 1 line from the input files, as those lines are not records.

How I defined a top tag? If a tag has at least 10 accepted answers in last 1 year for a user, then that tag can be a top tag for that user. If a user has more than 3 tags that satisfy top tag definition, I chose only top 3 based on accepted answers count sorted in descending order.

For each user, I identified the posts they made that were marked as accepted answers for corresponding questions and took tags of those questions. Then, counted how many times a tag is

appearing and sorted the results based on accepted answers count and took top 3 results for each user. Also, it is obvious that not every user will have top tags and each user can have at most 3 tags as top tags. I will store these results as a user to tags mapping in Cassandra database.



*Figure 9: Apache Spark setup on AWS Cluster*

For each user, I stored user DisplayName, UpVotes, DownVotes and Reputation in the Cassandra database. To calculate, trending tags in Stack Overflow, I took tag name and count values from tags.xml, sorted them based on the count to take top 10 tags and stored the results in the Cassandra database.

I have used Spark SQL to create temporary tables for each of the input files and wrote SQL queries to fetch information. All my batch layer code is written in Scala.

## 4.3 Data Ingestion

Kafka is used for ingesting new data to Spark streaming as well as HDFS. I have created 3 topics on Kafka cluster to store posts, UpVotes, DownVotes respectively. I have only 1 producer writing to each of these topics. Since I have not connected to Stack Overflow real time-API to get real-time data, I have saved 1000 records in a file and my producer is reading records from that file to write to Topics.

Command to create a topic:

```
kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 3 --partitions 1 --topic posts
```



**Instance type: m4.large**  
**Number of Instances: 3**

*Figure 10: Kafka, Flask Cluster on AWS*

posts are the topic name and 3 is the replication factor for my Kafka cluster setup. From Kafka topics, records are read by Spark Streaming for processing. My producer script is written in python.

## 4.4 Speed Layer

Spark Streaming acts as a producer for my Kafka cluster and reads records from Kafka topics. I have created different DStream objects for each of the record types like posts, Upvotes, DownVotes read from Kafka topic and operated on DStream object.

Whenever a new post record comes to Spark Streaming, I took title and tags of that post. For those tags, I got the users with those tags as top tags from the batch layer results stored in Cassandra. Now, I have the user to tags mapping and title to tags mapping, from which I have identified the user to title mapping and stored the results in the Cassandra database.

Whenever a new UpVotes or DownVotes record comes from the Kafka topic, I overwrite corresponding user counts in the real-time views in the Cassandra.

Spark and Spark Streaming are on same AWS cluster.

## 4.5 Cassandra

Cassandra database acts as the serving layer database, storing all the results coming from Batch and the Speed Layer.

I created a KeySpace named stackoverflow, followed by a list of tables to store the results:

1. userprofile - **id**, displayname, upvotes, downvotes, reputation
2. toptags - **userid**, tags
3. usertoquestion - **userid**, questions
4. trendingtags - **updatedon**, **count**, tagname

Spark Streaming connects to Cassandra to read and write the speed layer results.



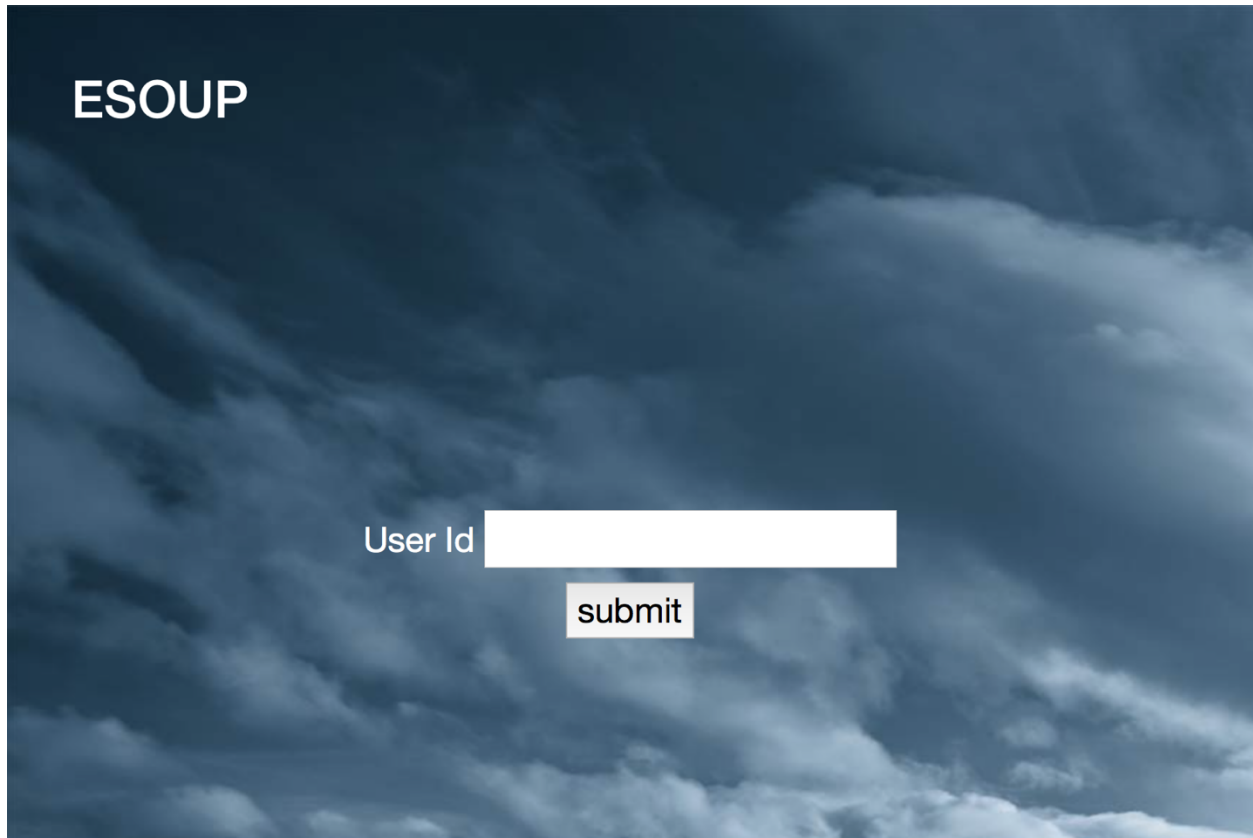
*Figure 11: Cassandra cluster on AWS*

# Chapter 5 - Results

I have run the batch layer on the master data set stored in HDFS. It took nearly 2 hours to process the data and stored results in the tables created in the Cassandra database.

- User records - 6 million approx.
- Users with top tags - 1789

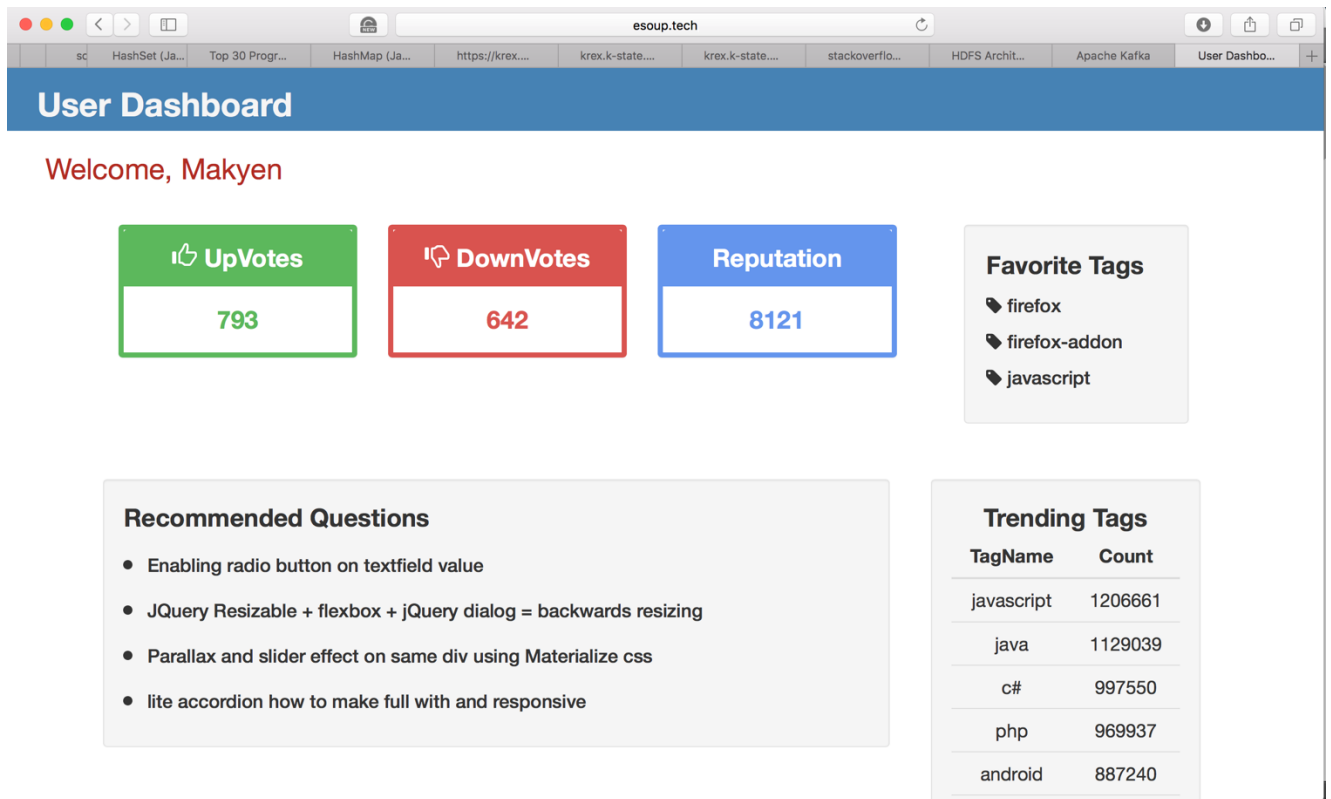
I have created an application to see how batch layer results are used and updated when some real time data comes into the application. Whenever a new post record comes to the speed layer, it is mapped to the users with those post tags as top tags and that post is recommended to User in User dashboard. Also, whenever new UpVotes record comes, the corresponding UpVotes value is updated immediately. I have load tested the application, it can handle 1000 records per second in real time.



*Figure 12: Application Home Page*

ESOUUP - Extending Stack Overflow Profile

Whenever a user enters the userid, he is redirected to user dashboard page. Use dashboard displays user performance numbers, recommended question, favorite tags and trending tags in the application along with their counts.



*Figure 13: User Dashboard Page*

All information shown in the dashboard gets updated in real-time if any event related to the corresponding happens in the background.

```
cqlsh:stackoverflow> select * from toptags limit 10;
```

userid	tags
733721	{'ruby-on-rails'}
4035472	{'c#', 'reflection'}
184595	{'adobe-analytics'}
3773011	{'firefox', 'firefox-addon', 'javascript'}
5182221	{'julia-lang'}
4243	{'bazel'}
1754112	{'android'}
1103681	{'rust'}
1295678	{'python'}
826057	{'c#'}

Figure 14: Top Tags

```
cqlsh:stackoverflow> select * from trendingtags;
```

updatedon	count	tagname
2016-10-06 06:16:20.877000+0000	1206390	javascript
2016-10-06 06:16:20.877000+0000	1128886	java
2016-10-06 06:16:20.877000+0000	997494	c#
2016-10-06 06:16:20.877000+0000	969739	php
2016-10-06 06:16:20.877000+0000	887060	android
2016-10-06 06:16:20.877000+0000	770046	jquery
2016-10-06 06:16:20.877000+0000	624041	python
2016-10-06 06:16:20.877000+0000	572584	html
2016-10-06 06:16:20.877000+0000	467189	c++
2016-10-06 06:16:20.877000+0000	457194	ios
2016-10-06 06:16:20.877000+0000	20	mysql
2016-10-06 06:16:20.877000+0000	13	spring
2016-10-06 06:16:20.877000+0000	9	swift
2016-10-06 06:16:20.877000+0000	5	user-interface
2016-10-06 06:16:20.877000+0000	3	tensorflow
2016-10-06 06:16:20.877000+0000	1	windows

Figure 15: Trending Tags

# Chapter 6 - Conclusion & Future Work

## 6.1 Conclusion

The Lambda Architecture provides a consistent approach to building a big data system that can perform real-time updates with low latency, high throughput and in a fault-tolerant way. A single tool cannot meet all the requirements of an application, so there is a need to use technologies/tools as a conjunction in a layered way as specified in the Lambda Architecture to achieve application that satisfies our requirements. Stack Overflow is a proper application to demonstrate the Lambda Architecture as it has huge user community with large posts getting created every second.

## 6.2 Future Work

There is scope for a lot of improvement in obtaining results with better accuracy. We can use some machine learning techniques in identifying top tags for a user. Also, tags are provided by a user which are not validated, building a prediction model that predicts accurate tags to a question can improve question recommendation in overall. We can also add Job Recommendations to the application based on user expertise. We can add events like MeetUps, Conferences information to the application based on user location and user expertise.

All this information keeps user more occupied whenever he visits the application and makes him an active user of the application. We can also increase the input events per second and batch dataset size to perform load testing on the application and scale the application accordingly.



# Bibliography

- [1] Apache Kafka. URL <http://kafka.apache.org/documentation.html>
- [2] Apache Spark. URL <http://spark.apache.org/>
- [3] Apache Cassandra. URL <http://cassandra.apache.org/>
- [4] Hadoop Architecture. URL [https://hadoop.apache.org/docs/r1.2.1/hdfs\\_design.html](https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html)
- [5] Tathagata Das, Matei Zaharia and Patrick Wendell. (2015). Spark Streaming Architecture. Retrieved from <https://databricks.com/blog/2015/07/30/diving-into-apache-spark-streamings-execution-model.html>
- [6] Nathan Marz and James Warren. (2015). Big Data Principles and best practices of scalable real-time data systems. Retrieved from <https://www.manning.com/books/big-data>
- [7] Data Source. URL <https://archive.org/details/stackexchange>
- [8] Flask. URL <http://flask.pocoo.org/>
- [9] Tornado Server. URL <http://www.tornadoweb.org/en/stable/>