Building a scalable distributed data platform using lambda architecture

by

DHANANJAY MEHTA

B.Tech., Graphic Era University, India, 2012

A REPORT

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department Of Computer Science
College Of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2017

Approved by:

Major Professor
Dr. William H. Hsu

Copyright

Dhananjay Mehta

2017

Abstract

Data is generated all the time over Internet, systems, sensors and mobile devices around us this data is often referred to as 'big data'. Tapping this data is a challenge to organizations because of the nature of data i.e. velocity, volume and variety. What make handling this data a challenge? This is because traditional data platforms have been built around relational database management systems coupled with enterprise data warehouses. Legacy infrastructure is either technically incapable to scale to big data or financially infeasible. Now the question arises, how to build a system to handle the challenges of big data and cater needs of an organization? The answer is Lambda Architecture.

Lambda Architecture (LA) is a generic term that is used for a scalable and fault-tolerant data processing architecture that ensure real-time processing with low latency. LA provides a general strategy to knit together all necessary tools for building a data pipeline for real-time processing of big data. LA builds a big data platform as a series of layers that combine batch and real time processing. LA comprise of three layers - Batch Layer, responsible for bulk data processing; Speed Layer, responsible for real-time processing of data streams and Serving Layer, responsible for serving queries from end users. This project draw analogy between modern data platforms and traditional supply chain management to lay down principles for building a big data platform and show how major challenges with building a data platforms can be mitigated. This project constructs an end to end data pipeline for ingestion, organization, and processing of data and demonstrates how any organization can build a low cost distributed data platform using Lambda Architecture.

Table of Contents

List of Figures . . . viii

Acknowledgements . . . ix

1 Introduction . . . 1

2 Background and related work . . . 3

2.1 Background . . . 3

2.2 Lambda Architecture . . . 4

2.3 Need for data supply chain . . . 8

3 Architecture . . . 9

3.1 Data supply chain management . . . 9

3.2 Building data supply chain . . . 11

3.3 Lambda Architecture . . . 12

3.3.1 Batch layer . . . 13

3.3.2 Speed layer . . . 17

3.3.3 Serving layer . . . 19

3.4 Data organization . . . 20

4 Implementation . . . 23

4.1 Overview . . . 24

4.2 Data . . . 26

4.3 Batch layer . . . 27

4.4 Speed layer . . . 28

4.5 Serving layer 29
4.6 Data organization 30
4.7 Infrastructure 32
5 Results 33
5.1 Performance improvement 36
5.2 Testing 38
6 Challenges 39
7 Conclusion and future work 42
7.1 Future work 42
7.2 Conclusion 43
Bibliography 44

List of Figures

2.1 Lambda Architecture overview 5
2.2 Layers in Lambda Architecture 6
2.3 Lambda Architecture diagram 7

3.1 Data supply chain 10
3.2 Components of data supply chain 11
3.3 Batch computation approaches 13
3.4 HDFS architecture 14
3.5 Spark library 15
3.6 Spark architecture 16
3.7 Kafka topic architecture 18
3.8 Spark streaming architecture 19
3.9 Visualizing CAP theorem 20
3.10 Cassandra data organization 21

4.1 Project use-case 23
4.2 Existing architecture 24
4.3 Data format 26
4.4 Batch processing pipeline 27
4.5 Time range queries 28
4.6 Data transformed by Spark 28
4.7 Stream processing pipeline 28
4.8 Speed layer pipeline 29
4.9 Different views for time based queries. 30

4.10 Cassandra schema 31
4.11 AWS infrastructure 32

5.1 User login 33
5.2 Trend analysis for previous year 34
5.3 Top 5 items during previous year 34
5.4 Top 5 items from last month 35
5.5 Details of individual item 35
5.6 Data organization for Platter 36
5.7 Visualizing query: weekly aggregate-1 37
5.8 Visualizing query: weekly aggregate-2 37
5.9 Visualizing query: top-k items 38

6.1 Existing vs new architecture 40

Acknowledgments

I would like to express my deepest gratitude to my adviser Dr. William H. Hsu of the Department of Computer Science, College of Engineering at Kansas State University. Apart from being an adviser he was also my mentor. His doors were always open whenever I needed some guidance or ran into a trouble about my work. He has consistently motivated me to pursue my research and career objectives. He has steered me in the right direction whenever he thought I needed it. I thank him for his constant input and feedback on this project as well as my Master's at Kansas State University.

I would also like to thank Dr. Mitchell L. Neilsen and Dr.Daniel Andresen for serving on my M.S. committee. Without their passionate participation this work could not have been successful.

Finally, I express my gratitude to my parents and brother for providing me their constant support and encouragement throughout my study. This accomplishment would not have been possible without them.

Chapter 1

Introduction

Data is produced all the time over Internet, systems, sensors and mobile devices around us. This volume, velocity, and variety of data produce huge data silos that are commonly referred to as 'Big Data'. It is expected that by year 2020, about 1.7 megabytes of new data will be created every second by every single human being on earth. Currently, users perform 40,000 search queries every second on Google, which makes it 3.5 billion searches per day and 1.2 trillion searches per year. Similarly, Facebook users send on average 31.25 million messages and view 2.77 million videos every minute. In next five years there will be 6.1 billion smart-phones and over 50 billion connected devices globally which indicate that there will be more data than ever before[1]. Big data empower users with various actionable insights such as creating personalized medicines based on patient genetic information, delivering customized solutions to users concurrently or analyzing billions of financial transactions. All these benefits come at the cost of handling big data.

Big data is a challenge for big organizations as their existing data platforms are based on legacy relational database management systems (RDBMS) coupled with enterprise data warehouses (EDW). With big data, it is realized that legacy infrastructure is technically incapable to scale up to volume and variety of data and financially impractical for storing and analyzing this data. Some of these challenges include ingesting various data sources, adding microservices, integrating different data formats, transforming data to fit existing

databases and data warehouses efficiently. Technologies evolved constantly but organizations adopted them in a piecemeal fashion due to lack of a concrete architecture that could knit these tools together. As data ecosystem became more complex, organizations are littered with data silos which remain underutilized.

Small organizations also face difficulty in handling big data but their challenges are slightly different when compared to big organizations. Start-ups start with a barebone infrastructure to get them off ground with their product. As their popularity increase the volume of data and traffic generated by their users increase as well. This require their product to be scalable and fault tolerant because early stage is very crucial for growth of these organizations. Any failure or downtime may impact their business and growth severely. These organizations face several problems such as low human resources, small budget, and lack of experience in addition from challenges that were faced by larger organizations.

The goal of this project is to build a scalable, fault-tolerant and cost effective solution for processing big data using Lambda Architecture. This project is based around a start-up company called Platter[1]. This project highlights the limitations of their existing architecture and challenges they anticipate in future. This project not only address existing limitations but also propose additional features that can be added to their existing product. Although this project is based around a small organization but, this solution can be incorporated by any other organization without much change.

This project draw analogy between modern data platforms and traditional supply chain management to lay principles of building modern data platform and demonstrate how data supply chain management can overcome challenges with building data platforms at these organization. Lambda Architecture(LA) provides a framework to knit together all necessary tools to build a scalable data system which minimize cost, complexity and failure. This project establish that a scalable and robust big data platform can be built based on the principles of a data supply chain implemented with Lambda Architecture.

[1] Original name has been changed.

Chapter 2

Background and related work

2.1 Background

Batch jobs have been traditionally running overnight to process data and generate results before next business day. These jobs generate reports and populate dashboards when they end. There was an obvious reason to run these jobs off peak hours i.e. overnight as they were resource crunching and took hours to process the data. Batch processing have evolved from standalone jobs to distributed processing on Hadoop. The term 'Big Data' was first coined by Roger Mougalas in 2005 which referred to a large set of data that is impossible to manage and process using traditional tools. The same year Yahoo created 'Hadoop' based on a paper "MapReduce: Simplified Data Processing on Large Clusters" by Jeffrey Dean and Sanjay Ghemawat from Google[2]. Hadoop is the first framework built to handle big data. Hadoop originally consisted of Hadoop Distributed File Systems(HDFS), Yet Another Resource Negotiator(YARN), and MapReduce. Over last decade Hadoop has evolved into an ecosystem consisting of several tools and frameworks that address specific problems associated with big data. They are broadly classified as Batch computation systems such as Hadoop, Spark; Serialization frameworks such as Avro, Parquet; NoSQL databases such as Cassandra, MongoDB; Messaging System such as RabbitMQ, Kafka; Real-time computation such as Flink, Storm. Time delay is increasing for batch processing with increase in data

size. One easy way to overcome this delay is to scale infrastructure horizontally but this is a very expensive solution.

Now, why is real-time processing needed? Because delay can lead to revenue losses or missed business opportunities for organizations. Lets consider a simple example of advertising campaign on mobile platform to understand how batch and real-time processing go hand in hand. Mobile ad campaigns usually run for short durations. Performance of these campaigns need constantly monitoring to adjust their priorities based on the feedback. Advertisers lose visibility if a campaign under delivers, and platform lose revenue if over delivered. Batch data can help advertisers choose target audience but it cannot be used to tune priorities for ads. This is when real-time processing is needed. Real-time systems continuously process data as it arrives and generate results. But real-time processing is very delicate as it is impacted by several factors such as delay in incoming data, system failure, out of sequence data, corrupt data and many more factors.

Both batch and real time components need to work in tandem to build a complete big data system. Lambda Architecture a perfect solution as it combine both batch and stream processing into a single framework.

2.2 Lambda Architecture

"Lambda Architecture" was first coined by Nathan Marz as a generic term that refer to a scalable and fault-tolerant data processing architecture that ensures real-time processing with low latency. The main idea behind Lambda Architecture (LA) is to build a big data system as a series of layers. LA comprise of three layers - Batch Layer, for bulk data processing; Speed Layer, for real-time processing and Serving Layer, serve results to end users[3]. Each layer satisfies "subset of" property as it is built upon the functionality provided by the layer beneath it. Each layer is subset of entire big data system. LA build a robust system which is fault-tolerant against both hardware failures and human errors by abstracting functionality and complexity to individual layer.

Traditionally data has been transformed to fit into the database but today events is data.

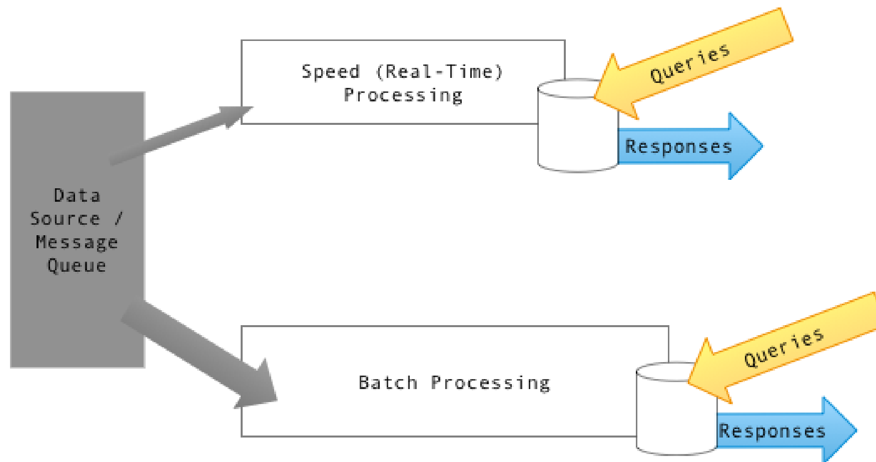

Figure 2.1: *Lambda Architecture overview*[4]

Data needs to be stored in the form it is generated hence it is immutable. Let's consider a situation where an organization is looking to analyze one Terabyte dataset on fly. The queries would look like this:

$$query = function(all\ data)$$

If this query was ever runs it will take hours before any result. Lambda Architecture provide an alternative where this query can run against a precomputed view. The query would now look like this:

$$batch\ view = function(all\ data) - Step1$$

$$query = function(batch\ view) - Step2$$

Batch view precomputes results for possible queries. Now, instead of computing the query on fly, results can be derived from a precomputed batch view. Batch view avoids processing entire dataset each time. But, generating a batch view is time intensive operation and lot of new data gets created during this period. A real-time system cannot allow this latency

between start and end of batch processing as queries will be out of date.

$$query = function(batch\ view) - recent\ data$$

Therefore, real-time view is required

$$real - time\ view = function(recent\ data)$$

Lambda Architecture resolve this problem by combining batch and real-time view. Lambda Architecture comprise of three layers:

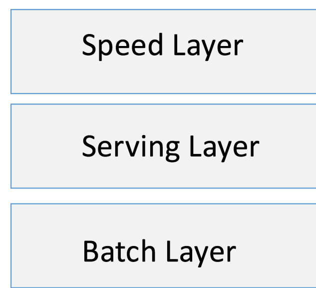

Figure 2.2: *Layers in Lambda Architecture*

1. **Batch Layer** - This layer process bulk data from master dataset and computes batch views. The batch performs two major tasks: organize master dataset, and generate batch view. Batch computations tools such as Spark or Hadoop perform processing in this layer. This layer has high latency due to volume of data processed in this layer. Batch layer can be scaled horizontally to reduce latency. Any new data coming into system is appended to master dataset.

2. **Speed Layer** - There is a lag between actual view and view produced by batch layer which is equivalent to time lag between starting batch processing and getting the results. A true real-time system cannot allow lag in views, therefore data gathered while computing batch view needs to be processed in parallel. Speed layer ensures that there is no latency in views presented to user for any query. Speed layer perform incremental computation instead of recomputing complete data.

3. **Serving Layer** - Once batch views are generated they can be queried by serving layer. Serving layer allow random reads but random writes or updates are not allowed. Random writes bring complexity and reduce performance of a database. By not supporting these operations, databases in serving layer stay simple, robust, easy to configure and operate. Although it takes few hours for new data to propagate through batch layer into serving layer but it satisfy properties desired in a Big Data system i.e. minimum read latency and maximum accuracy. Delay in generating batch view is countered by views generated by Speed Layer.

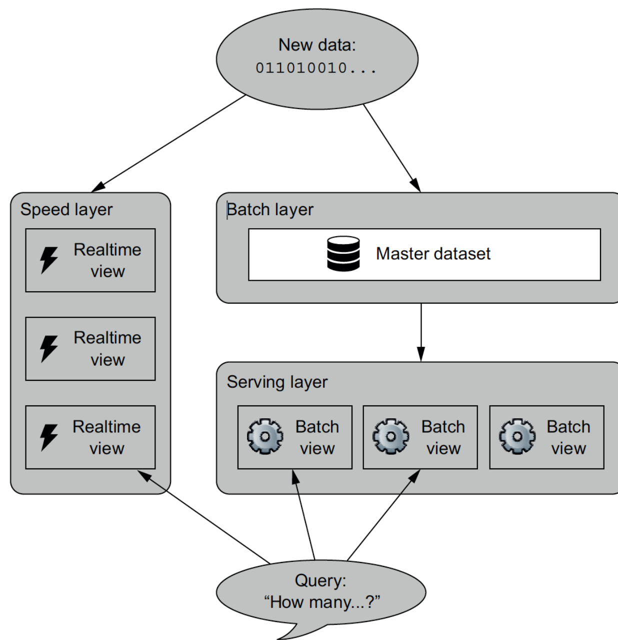

Figure 2.3: *Lambda Architecture diagram*[3]

Therefore, when a user queries serving layer the query is resolved by looking at both batch and real-time views. This ensures minimum latency and most up to date results. This architecture can be summarized in terms of following three functions:

Batch Layer : $Batchview = function(masterdata)$

Speed Layer : $Real - timeview = function(real - timeview, newdata)$

Serving Layer : $query = function(batchview, real - timeview)$

Lambda Architecture isolate complexity of entire system to each individual layer. This not only help to make system simpler but also reduce possible errors and help in failure recovery. Speed layer is far more complex compared to serving or batch layer as it involves random reads, writes or updated. Once data is processed by the serving layer, real-time views are no longer required and can be discarded. If anything ever goes wrong with speed layer entire data can be discarded and system can start fresh without any loss.

2.3 Need for data supply chain

As data grows bigger it becomes more difficult to manage and analyze this data. In order to construct a big data architecture, it important to manage and process data. Goal of a big data solution is not to create a giant factory to process all the data thrown onto it. Big data platform needs to be created such that is can accepts data from a variety of sources, processes the data, transform the data, store the data, and use the data wherever or whenever needed. Big data does not necessarily need massive data warehouse as it has been the case traditionally. In fact, big data need a design that neatly organize these data silos and use them based on the need and importance of data. Quality and not the quantity of data is important when it comes to generate insights from data. Quantity of data does matter but it needs to be made available for use as quickly as possible when needed. Variety of data is also an important factor when it comes to decision making based on data.

The volume of data is bound to rise for any organization but there is significant danger of diluting focus due to different tools and data available. This make it really important to design system considering objectives and challenges faced. Supply chain management (SCM) is a framework to coordinate materials, information, and finances as they move from supplier to manufacturer to wholesaler to retailer to consumer. Big-data platform is similar to a SCM as it involves different stages and different entities. My design based on data supply chain therefore address this challenge of building an optimized big data solution.

Chapter 3

Architecture

Lambda Architecture is a design method that builds data platform by combining different tools and technologies into a real-time scalable and fault-tolerant system. There are several components that build together a data platform. Therefore, before diving deep into Lambda Architecture (LA) let us first understand various components of a data platform. As mentioned in section 2.3 this project draw parallel between data platform and supply chain management. This analogy will help to correlate problems faced in building a data platform in an organization and how they can be resolved by correlating to a well-established process of supply chain management (SCM).

3.1 Data supply chain management

SCM involves procurement of raw materials, storage of that material for easily availability in inventory, work process to make goods, and finished goods for consumption. SCM is a interconnected network of stages from point of origin to point of consumption. Each stage is built on previous stage and finished product is delivered to the end customers. Data platform is analogous to a SCM where data sources can be considered as source of raw data sources, data storage can be considered as warehouse, processing frameworks can be considered as work process, and analysis from data can be considered as finished goods. LA is the strategy

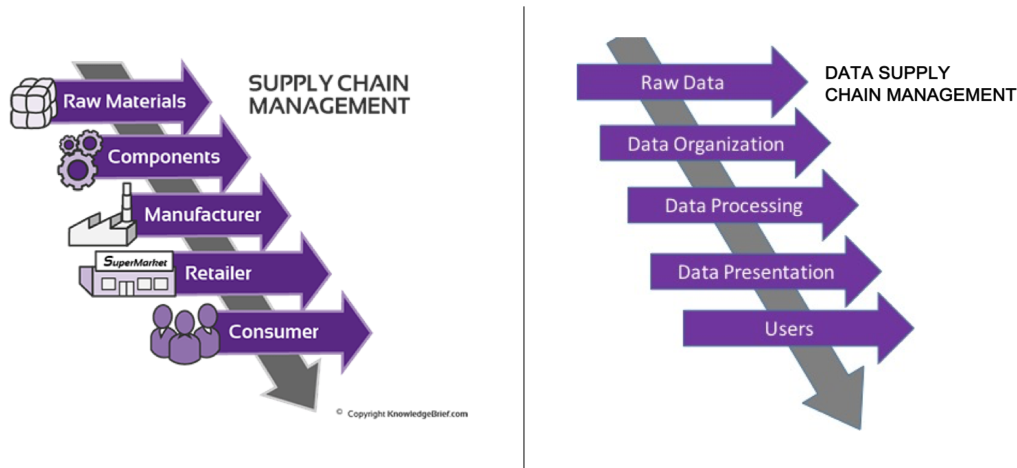

Figure 3.1: *Modern data platform vs supply chain management*[5]

that help to put this SCM in place for a data platform. There are three major challenges that a data supply chain can help to address[6].

1. **Data movement:** Data has been traditionally organized in relational tables in organizations. Moving this bulky data is time consuming but a straight forward process as data is well formatted and in uniform shape. The challenge with big data is the volume, velocity and variety with which data is generated. Data needs to be tapped from multiple data sources without loss. Having a modern data infrastructure that can collect relevant data can help to deliver better insights. A data supply chain helps to manage data movement problem by enabling multiple channel data input and storing this data without mutating it. This data is stored for fast read and high write throughput.

2. **Data processing:** Organizations have transformed data to fit their database systems in past. They are accustomed to processing data which is normalized and well organized across relational tables. There is need for new tools and a nascent architecture to put together different tools to process big data. Let's consider another example of mobile ads. Businesses can be delivered promotions or discounts to customers' devices when they are near a likely place of purchase by monitoring a customers location. To make this more intelligent real-time data can be combined with historical purchase data for

customers. The promotions now delivered are tailored for same customer and increase likelihood of a purchase. In this example, batch data was processed with real-time data.

3. **Interactivity:** Data was organized in relational tables for faster access which made it easy for people to submit SQL queries and get results. The sheer volume of data induce a latency from minutes to hours to generate results on any such queries. Longer a user waits, longer it takes to get results from data. This situation gets worse if any critical decisions are based on the results. A data supply chain enables faster results by ensuring that the queries results in shortest time delay.

3.2 Building data supply chain

There are several components of a data platform that include *Data Ingestion, Data Organization, Data Processing, Data Analysis and Data Presentation.* These different components combine together to churn big data to generate end results. Big data can be analyzed in different dimensions so an organization shall set up core objectives of building a data platform. These major components of a data supply chain have been described below in more details:

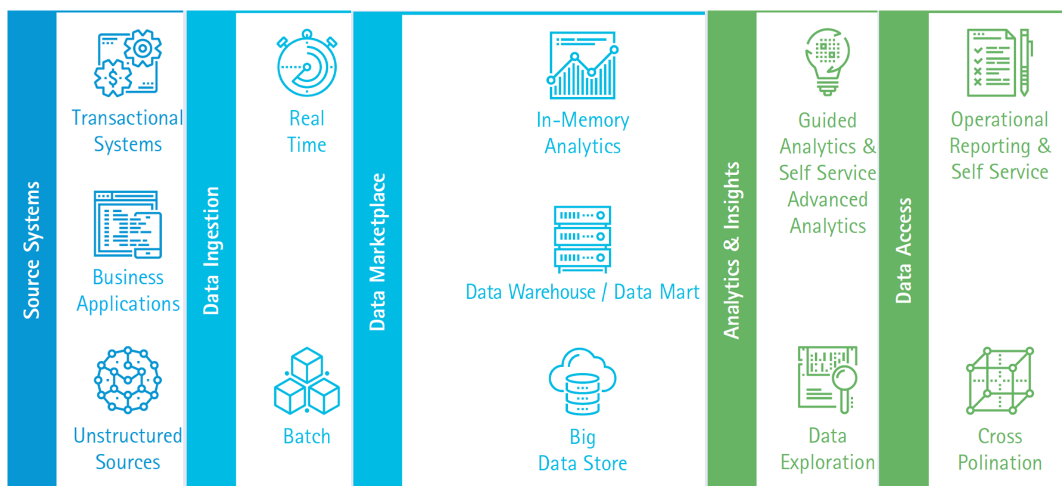

Figure 3.2: *Components of data supply chain*[6]

1. **Data ingestion:** This component is responsible for bring data to supply chain from different source systems. Data can be gathered in real-time or offline. Data source might include legacy systems, traditional data storage and warehouses, social media, or device data. Data can come from logs, API, data dumps, streams and many other ways. Some popular tools for data ingestion include Apache Sqoop, Apache Flume or Apache Kafka[7].

2. **Data organization:** This component is responsible for cleaning, pre-processing, transforming and storing raw as well as processed data. This layer ensures low read latency and high write throughput. Data gathered is stored in an immutable form because if there are any mistake during processing all data will be destroyed[8]. This ensure much stronger human-fault tolerance guarantee than traditional systems. Some popular tools for data organization Apache Cassandra, Apache Impala, Apache Kudu[7].

3. **Data processing:** This component is responsible for processing data and generate goods to be consumed by end users i.e. it generates end results that can be queried by the end users. Data can be processed in batch mode or real-time mode. Some popular tools for data processing include Apache Spark, Apache Hadoop, Apache Storm[7].

4. **Data insights:** This component provides results for user queries. It supports information presentation, analysis, and advanced analytics methods to users. Some popular for data insights include Tableau, SAS, R.

3.3 Lambda Architecture

Data Supply chain set principles for building and managing a data pipeline. These principles can be realized by LA. LA implement all the components of a data supply chain. LA has three layers - **batch layer, speed layer, stream layer**. Batch Layer focus on ingestion, storage and processing of batch data; speed Layer focus on ingestion, storage and processing of stream data; serving Layer focus on retrieval of views loaded by the batch and speed layers.

3.3.1 Batch layer

Data can be ingested from multiple source systems. LA store data in a master dataset which form core of the architecture and serve as the source of truth. Even if all data was corrupted or lost in other layers, master dataset can reconstruct the rest of the lost data. This is possible because views served by serving layer are generated by processing master dataset and speed layer is formed by data streams. This make LA fault tolerant to human errors and safeguard against any form of failure or corruption. As master dataset is usually too large to fit on a single machine, it is distributed across multiple machines in a cluster. As data is split across machines distributed file systems is needed to read, load or process this data[9]. Batch layer run various functions on master dataset to create a precomputed batch view. There

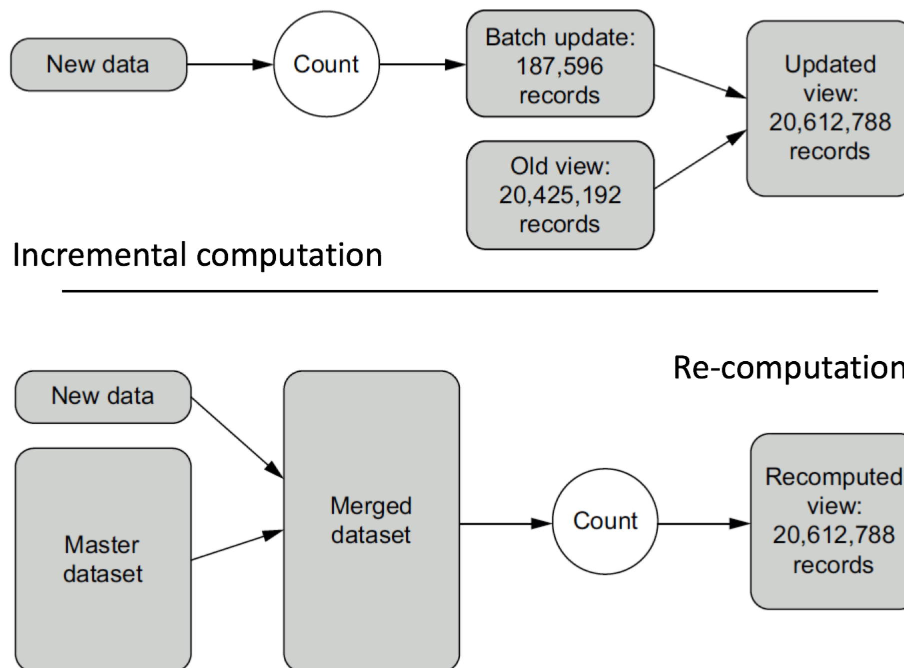

Figure 3.3: *Difference between incremental and re-computation*[3]

are two possible approach for computation in batch view: Re-computation and Incremental Computation. Re-computation scrape views generated during last run and recompute all over again over entire master dataset. Incremental computation updates existing batch views and only compute over new data. Incremental computation appears a simple, efficient and obvious solution but there are trade-offs in choosing the computation methodology.

Key trade-offs between the two approaches are performance, human-fault tolerance, and the generality of the algorithm. Once computation approach is decided batch processing tool such as Spark or Hadoop can perform the computations. The key paradigm behind distributed processing is MapReduce. MapReduce is a distributed computing paradigm that provides scalable and fault-tolerant batch computation.

Once computed data is stored into tables that is serve as view for serving layer. Section 3.4 provide more details on data organization. I used Hadoop Distributed File System (HDFS) and Apache Spark for implementing batch layer in my project.

Hadoop distributed file system (HDFS)

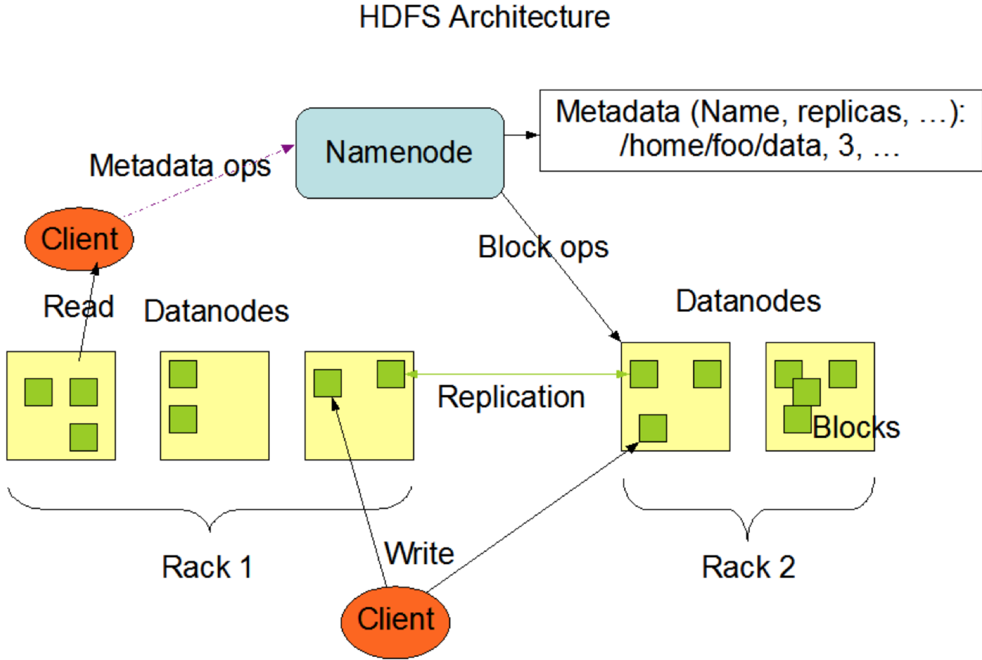

Figure 3.4: *HDFS architecture*[9]

HDFS is a fault-tolerant, distributed and scalable filesystem for storing and processing large dataset. HDFS manage data stored across clusters for computation by batch processing tools such as Hadoop or Spark that are deployed across a cluster. HDFS has a master/slave architecture with two types of nodes: a single NameNode and multiple DataNodes. NameNode act as master server managing file system namespace and regulate access to files

by users. DataNode manage storage attached to nodes that they run on. NameNode executes operations like opening, closing, and renaming files and directories. It also determines mapping between blocks to DataNodes. DataNodes are responsible for serving read and write requests from file system. When a file is loaded to HDFS it is first split into fixed size blocks between 64 MB and 256 MB. NameNode tracks file-to-block mapping and where each block is located. When any application processes a file stored in HDFS, it first queries NameNode for block location. Once location is known, application contacts the DataNodes directly to access the file contents. Each block is then replicated across multiple DataNodes that are randomly chosen also NameNode periodically receives a heartbeat from DataNode as an acknowledgment that DataNode is alive and working.

Apache Spark

Apache Spark is a distributed cluster computing framework[10]. What distinguish Spark from other frameworks such as Hadoop is in-memory data processing and speed. Spark provide a

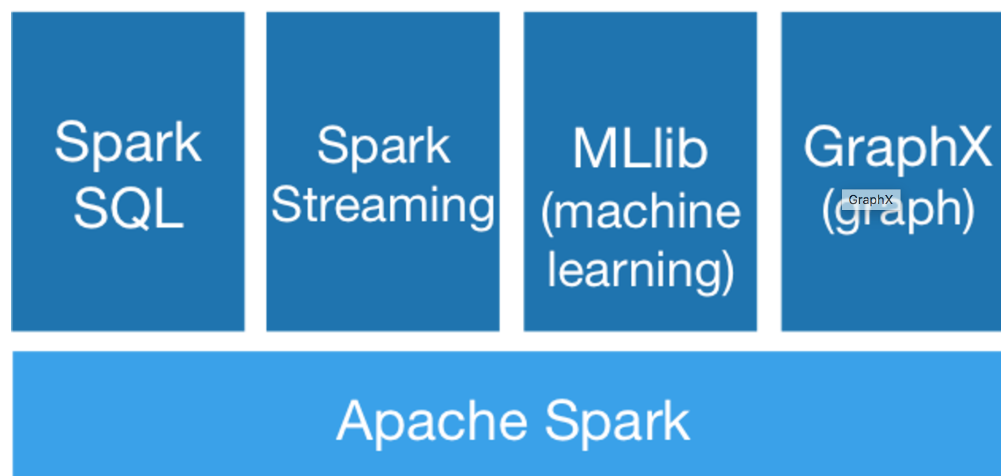

Figure 3.5: *Spark library*[11]

diverse library for data transformation, analytics, machine learning and graph processing on a large scale data in Batch mode or stream processing. Hadoop implements MapReduce in two-stage disk-based computations whereas Spark perform most computations in memory, and hence provides better performance than Hadoop in most of the cases. Spark can run locally or in clusters, or in cloud. It can run on top of Hadoop YARN, Apache Mesos,

standalone or in the cloud. Spark can access data from diverse sources such as HDFS, Cassandra, HBase, S3 etc[10]. Spark has a master/worker architecture where master node manages worker nodes which execute the jobs and functions. Apache Spark offer different libraries that include Core Spark, Streaming, SQL, MLlib, GraphX.

Spark Core is the heart of Spark and underlying engine for the library. It is responsible for task scheduling and providing efficient abstraction for in-memory cluster computing called Resilient Distributed Dataset (RDD). RDD is core of Spark which is designed to support in-memory data storage and distributed processing across a cluster in a fault-tolerant and efficient manner. RDD achieve fault tolerance using a directed acyclic graph (DAG) that

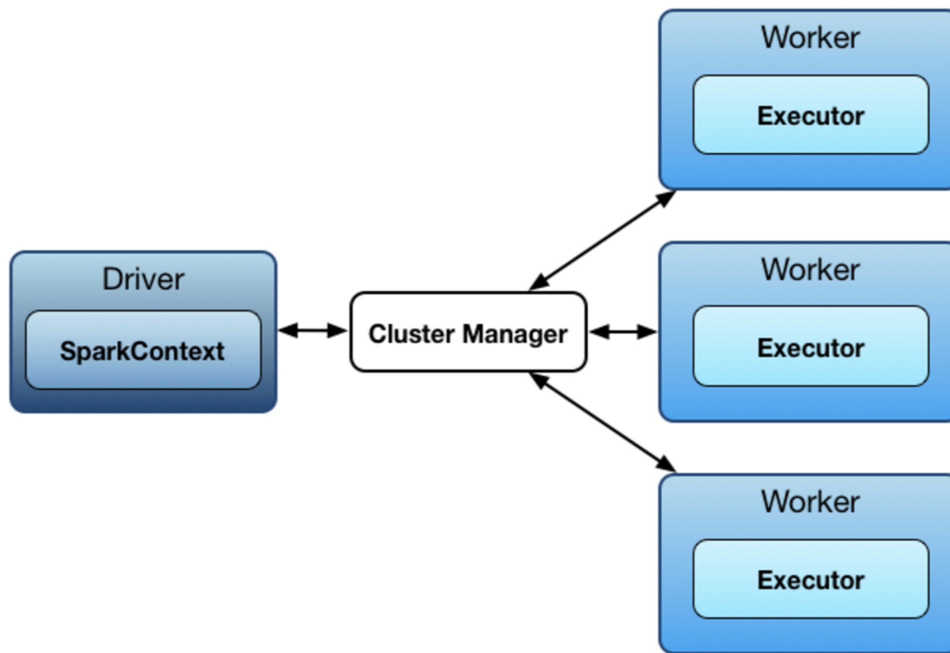

Figure 3.6: *Spark architecture*[12]

track lineage of transformations applied to data during processing. Efficiency is achieved in RDD through parallelization of processing across multiple nodes in cluster, and minimization of data replication between the nodes. Two basic operations take place once data is in RDD: Transformation and Action. Transformations are operations such as mapping, filtering, etc. which create a new RDD by changing original RDD. Actions are events such as count, show etc. that do not transform the data but they initiate the transformations to generate the

results. Transformations are lazily evaluated which means RDDx is not transformed to RDDy until any action is invoked. Lazy evaluation greatly improves spark performance as it avoids need to process data until all operations are done. Also, it avoids processing bottlenecks and deadlocks while waiting for a processing in a distributed environment RDD further boost performance of cluster as they remain in memory avoiding recomputation during iterative queries or processes.

Spark SQL is best suited for working with structured data. It leverages SQL query engine to big structured data allowing more complex analytics on data. Spark SQL supports JDBC, ODBC and Hive connections. Spark SQL can be easily integrated with existing databases, data warehouses and business intelligence tools. Spark MLlib implement commonly used machine learning and statistical algorithms such as classification, regression, clustering, PCA etc. on a distributed platform. Spark GraphX graph computation engine enable users to interactively build graph structures at scale. It supports analysis of and computation over massive graphs of data.

3.3.2 Speed layer

Speed layer produce views for data that was not processed while batch layer was processing master data. Speed layer encapsulate complexities around processing stream data. It provides low-latency real-time views required by the system. I used Kafka and Spark Steaming for implementing speed layer in my project.

Apache Kafka

Apache Kafka is a distributed message used to build a reliable and fault-tolerant streaming framework[13]. Kafka use a language independent TCP protocol to build a highly scalable publisher-subscriber (pubsub) model partitioned across a cluster. Kafka provide a core abstraction for data streams called 'Topic' to which records are published. Kafka Topics allow zero, one, or many consumers to subscribe to these topics. Kafka topics are multi-subscriber so they retains data even if its read by a process. This make Kafka more reliable

and fault-tolerant. Kafka has three main components - Producers, Consumers, and Brokers. A Producer is responsible for pushing data to one or more Kafka topics. Producer can either create data or act as end point of a data source that publish the data onto topics. Brokers are Kafka servers existing on separate nodes of a cluster that are responsible for all read/write actions. Each server is responsible for a pool of partitions with one node acting as "leader" while remaining nodes acting as "followers". When leader node fails, follower node is chosen as replacement which is replica of master node. To maintain fault tolerance partitions are replicated across multiple cluster nodes. Consumers are responsible to read the data from topics that was published by a Producer. A Consumer follows the publisher-subscriber model and reads only from those topics for which it is subscribed. In my project Spark Streaming acted as a consumer. The diagram below demonstrate how Kafka pubsub model works.

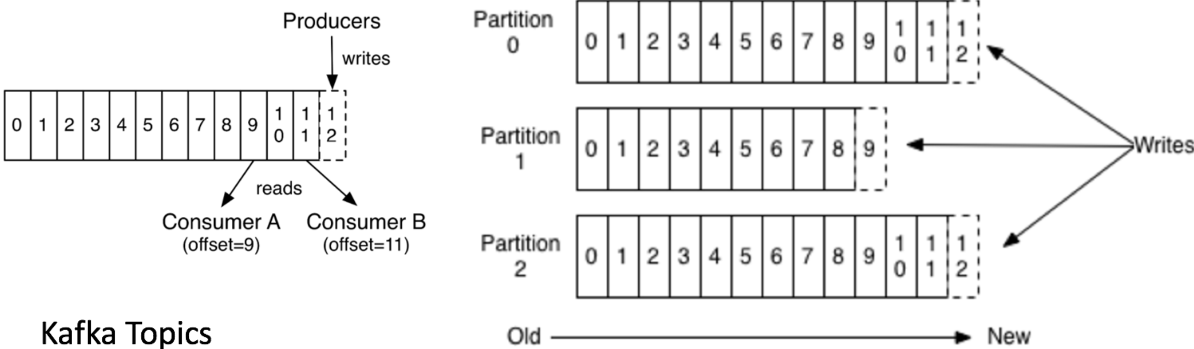

Figure 3.7: *Kafka topic architecture*[14]

Spark Streaming

Spark Streaming is responsible to process streaming data which can take input from different data sources such as HDFS, S3, Kinesis, Flume or Kafka. Unlike Flink, Spark Streaming is not a native streaming framework and therefore it does not process the streams based on an event. Rather, it uses an abstraction called discretized stream or DStream that represent continuous stream of data. DStreams can be created from input data sources. A DStream is internally represented as a sequence of RDDs. Using RDD for stream processing can

allow batch analysis of streaming data with little or no modification. The processed data is eventually pushed out to filesystems, databases, or dashboards.

Figure 3.8: *Spark streaming architecture*[11]

3.3.3 Serving layer

This layer serves views to users based on data generated by batch computation. The layer is the last component of batch section of LA. This is a read only view which is tightly coupled to batch layer as batch layer is responsible for updating the views for serving layer. Due to high latency of batch computations, serving layer views are usually out of date. This is compensated by views generated by speed layer which handles any data not yet available in batch computation. An application is built to handle thousands of simultaneous uses. Serving layer is distributed across the cluster for scalability and load balancing in order to support simultaneous access. Two factors are need to be considered when designing indices for serving layer - throughput and latency. Latency is the time required to answer a single query whereas throughput is the number of queries that can be served within a given time period. Therefore, databases used in serving layer are required to have certain features such as batch write to allow complete swap of old version and adding a new version of data, scalability to handle data volume, random readability to handle multiple reads from serving layer and fault tolerance. In this project I used Cassandra as a database for my serving layer queries. Cassandra is an open source distributed database which satisfies all the necessary features of a database for serving layer as listed above. More details about Cassandra are presented in following section.

3.4 Data organization

Workload and analytics get more complex with increase in size of data. The biggest challenge in building data platform is to efficiently and effectively organize available data Scalability and Distributed nature of database are not enough to achieve high performance. To determine data storage, one must consider how data will be written and how it will be read. Analyze nature of data and type of queries that can be requested from the serving layer. This not only help to optimize storage and organization of data but also influence query performance. Storage in distributed systems is guided by CAP theorem, also known as Brewer's theorem. It states that it is impossible for a distributed system to provide all three of the following guarantees simultaneously: Consistency (C) - all nodes see same data at same time. Availability (A) - every request receives most recently updated record in response. Partition Tolerance (P) - ability to handle downtime or failure of a node[15]. Consistency

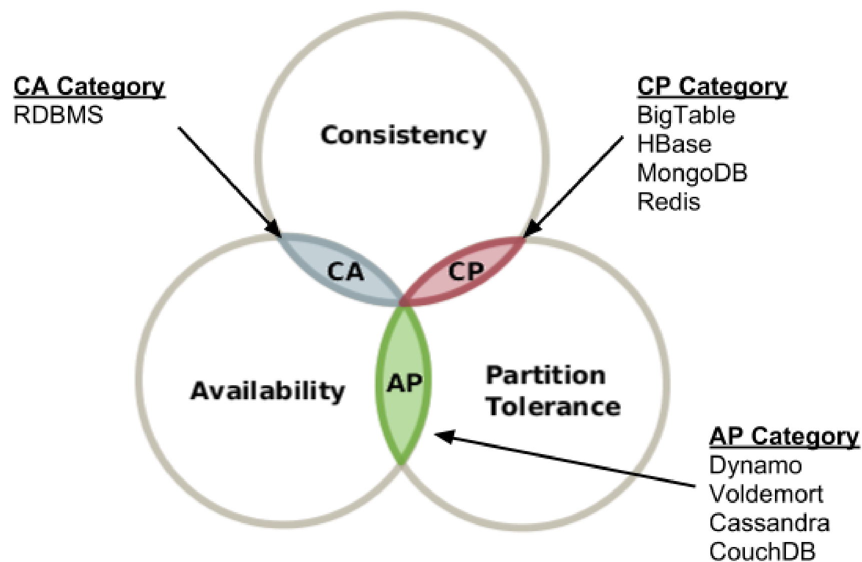

Figure 3.9: *Visualizing CAP theorem*[16]

is achieved by updating data on all the nodes before allowing new reads. Availability is achieved by replicating data and avoid any bottleneck or downtime. Partition Tolerance imply that failure of a part or complete system is not possible and achieved through various fault tolerance mechanism. Systems that allow reads before updating other nodes guarantee

high A whereas systems that update all the nodes before taking new reads will ensure C. P is a mandatory feature of database if it is distributed. There is a wide range of databases that can be used to build data platform based on specific feature needed for the platform. For example, Postgres or MySQL for relational data, Redshift for columnar storage, Redis or memcache for key value storage, Cassandra or InfulxDB for time series storage, CouchDB or MongoDB for document storage etc. For my project I have used Cassandra for storage.

Apache Cassandra

Cassandra is an open source distributed database that provide high scalability and fault tolerance. Cassandra is an AP system which make it suitable for systems that need availability more than consistency. Cassandra has a masterless architecture which means it does not have a single point of failure as all nodes are replicated. Every node in the cluster is identical and has similar role allowing high availability and scalability as any node can serve the request. Cassandra provides automatic data distribution across nodes in the database

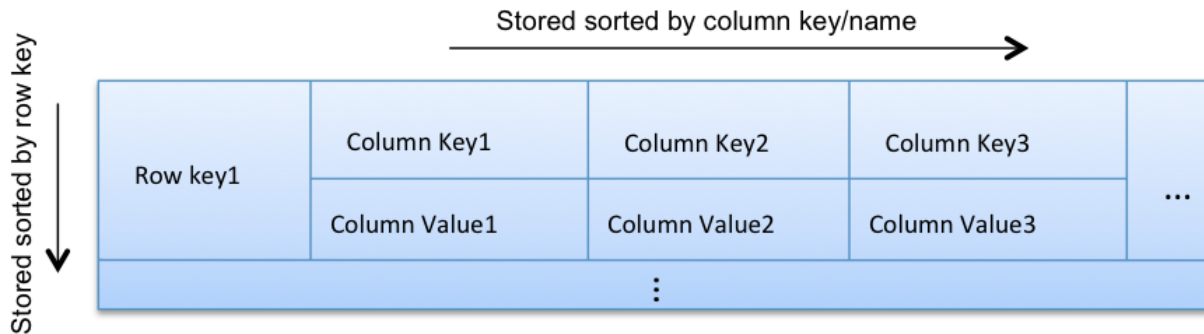

Figure 3.10: *Cassandra data organization*[17]

cluster. Also Cassandra replicate data across one or more nodes which means that if any node in a cluster goes down, data request can be processed by another node. This makes Cassandra highly scalable and fault-tolerant. Although Cassandra is not consistent it implements eventual consistency using "Gossip" protocol, a peer-to-peer communication protocol where nodes periodically exchange the information about themselves and other nodes and update nodes which are not in sync. Cassandra has a column-oriented data model that do not require a schema. As each row is not required to have same set of columns, columns can

be defined at any stage. Cassandra consist of keyspaces which are analogous to databases, column families which are analogous to tables in the relational model, keys and columns. Cassandra Query Language (CQL) is similar to SQL and used to change data, look up data, store data, or perform other functions. Cassandra possess all the features needed for serving layer as mentioned in section 3.3.3 and therefore it makes it a good fit.

Chapter 4

Implementation

The goal of this project is to build a cost effective, scalable and fault-tolerant big data platform based on the principles of data supply chain management built using Lambda Architecture (LA). The use case for this project was a small start-up organization, Platter. The objective was to address limitations of existing architecture and scale existing platform for future needs. Minimizing the cost is important factor which motivated use of open source tools that I described in Chapter 3. Section 4.1 provide more details on this use-case, problem statement, and the work that was done for this project.

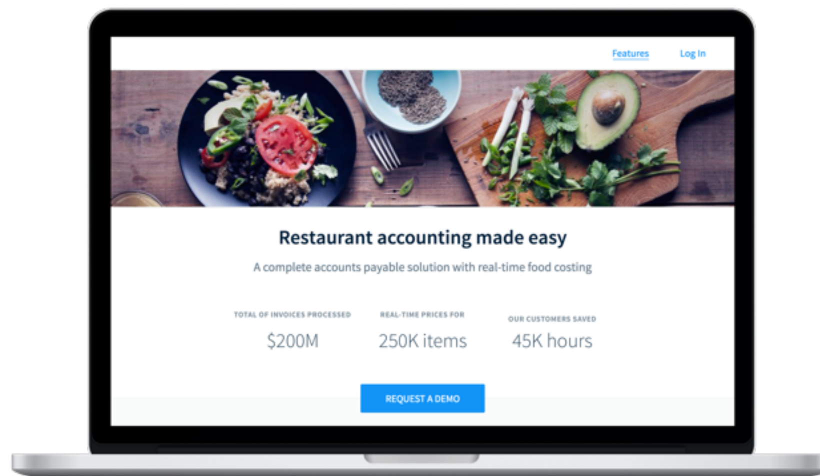

Figure 4.1: *Project use-case*

4.1 Overview

At present, restaurants process all of their paper invoices manually which is painful and expensive. This not only takes time but it is prone to frequent errors, and lacks an overall insight. Platter help restaurants to automate manual entries of paper invoices by offering a mobile application that allow restaurants to scan paper invoice. An inbuilt OCR in the application then digitize the records and store data on a relational table. In addition to this Platter also keep track of prices paid by a restaurant to buy food items, analyze these expenses, and present an overview of these expenditures to the restaurants. User can login to their online dashboards and can see the trends for requested period.

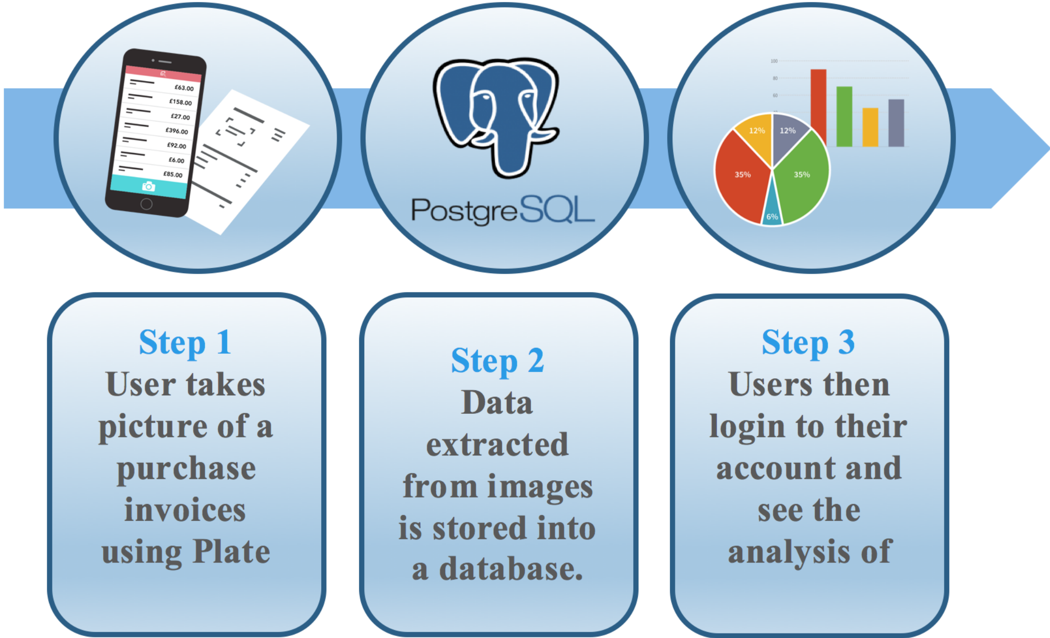

Figure 4.2: *Existing architecture*

So, what are the challenges with current implementation? What challenges it might face in future? I have listed these shortcoming and challenges below -

1. Users dashboard is not pre-populated and they have to request data to be displayed to dashboard for anytime period. This makes the dashboard less interactive.

2. Users queries are not pre-calculated hence each query is run on the fly taking time to

generate results and visualizations. This problem will grow as the data size increase.

3. Users can query for different food items they purchased over different timeframes. Users can simultaneously query for different items making it more difficult.

4. No feedback is provided to user in terms price variation and any significant change when a user scans the invoice.

5. Every time a user scan invoices the data is processed together with historical data and no instant update given on recent input.

6. Platter currently handle limited data sources; it does not get data from external sources such as Market Prices APIs. This restricts analytics capabilities as there are limited features in data.

7. Platter cannot scan and process high volume of invoices immediately as the data has to move to data warehouse before any feedback.

Building a distributed data platform using LA can address all these challenges.How?

1. **Batch layer** preprocess user data for different time frames to compute common views for example year, month, week or days. This avoid recalculating certain views or querying on fly.

2. **Speed layer** will process scanned invoices in real-time providing instant feedback to users. It will generate results for recently purchased items by a user.

3. **Serving layer** will take user queries and build the results based on the views generated by the batch layer and speed layer. This will provide an interactive dashboard to user by pre-populating user dashboard for common views, it will reduce latency for any query. Time complexity of the queries with new LA will remain constant even after huge data volumes.

4.2 Data

Platter provided a small dataset with nearly hundred thousand rows. This data was set of records scanned using Platter's mobile application. This sample data had records for ten restaurants and four hundred items purchased over three months. Data record had following attributes:

id	date	item_id	price	restraunt_id
1	2017/01/21	73	2.094	8
2	2017/01/21	96	1.243	8

Figure 4.3: *Data format*

- **id** - id of the record
- **date** - date of the invoice
- **item_id** - id of the item purchased by restaurant
- **price** - price of the item purchased
- **restaurant_id** - id of the restaurant that purchased the item

LA need data for batch and speed layer. As the sample dataset was very small data was simulated for both the layers. For batch layer, data was simulated for ten restaurants and five thousand items spread over three years generating one million unique transactions. This data was saved in .csv file format and stored over Amazon S3. For Speed layer, Kafka producer read data from a .csv file in a loop with random delay injected to mimic a real world scenario. This project generate following views for a user:

1. Yearly/Monthly Analysis, e.g. top-K items by Price Paid and Quantity Purchased
2. Analysis of individual items purchased by a restaurant.
3. Latest Items purchased by a Restaurant.
4. Statistics on item such as average, standard deviation, min, max, % change.

5. Trend in purchase, payments, or expense over year/month.

6. List most volatile items. (Note: volatility indicate % change in prices)

7. Latest items purchased by the Restaurant.

4.3 Batch layer

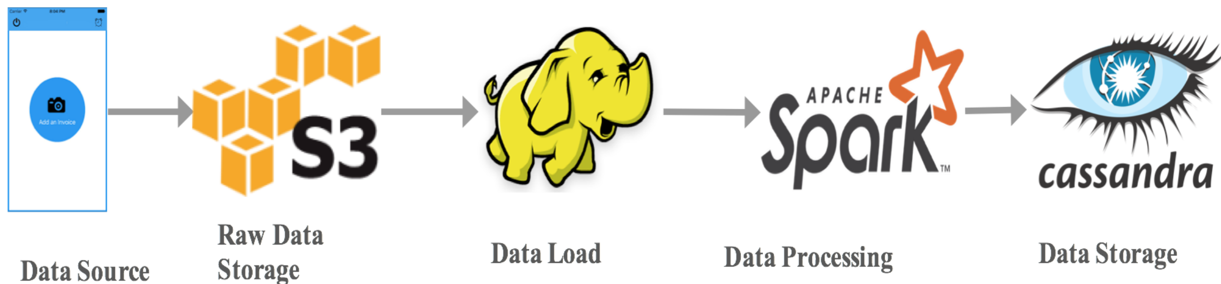

Figure 4.4: *Batch processing pipeline*

Fig 4.4 represent data pipeline for batch layer. Batch layer perform computation on the entire corpus of data to generate views listed in section 4.2 for serving layer. Data was stored on S3 and loaded onto HDFS. From HDFS data is read by Spark for processing. I am using pySpark for the project. Data is parsed by spark onto Spark dataframes. As data is structured I used Spark SQL library to perform most of the computations. The challenge here was that computations had to be performed for items purchased by restaurant over different time periods such as year, month, or day based on view to be generated. I split the date column into year, month, and day to generate these different views. To calculate statistics on data I used pyspark.sql.functions library. This library provides various functions such as **pyspark.sql.functions.stddev()** to find standard deviation, **pyspark.sql.functions.min(), pyspark.sql.functions.max()** to find minimum or maximum values, **pyspark.sql.functions.avg()** - to find average.

To provide top-K items purchased by any restaurant in a given time period I calculated the sum of price of the items purchased and sum of the quantity of the item purchased and aggregated while presenting results to end user. This was done using **pys-**

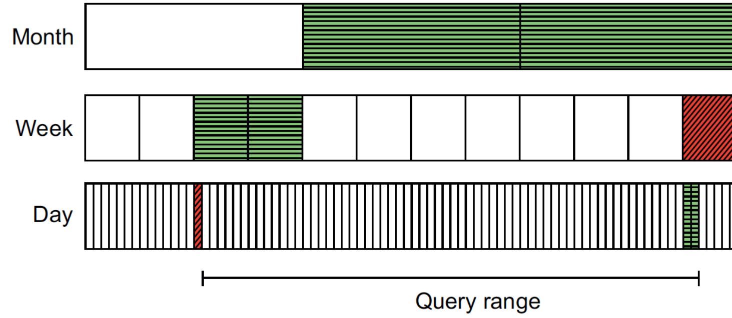

Figure 4.5: *Time range queries*[3]

id	date	item_id	price	restraunt_id	year	month	day	julian_day
1	2017/01/21	73	2.094	8	2017	01	21	21
2	2017/01/21	96	1.243	8	2017	01	21	21

Figure 4.6: *Data transformed by Spark*

park.sql.functions.sum() and **pyspark.sql.functions.count()** methods. After computing the sum for all items purchased or total sum spend by the restaurants the result is sorted and top-K items are inserted into the Cassandra table. The organization of data in Cassandra table will be presented in section 4.6.

4.4 Speed layer

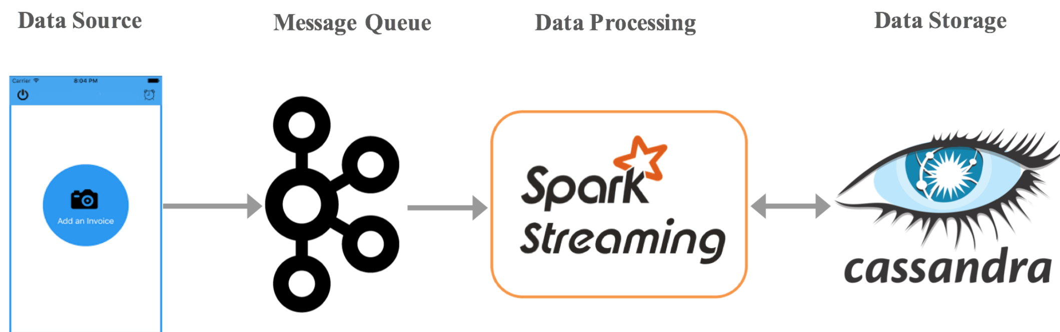

Figure 4.7: *Stream processing pipeline*

Figure 4.7 represent data pipeline for Speed layer. Speed layer serve views for real-time processing of invoices that are scanned by users and give feedback on items purchased

by them. Data is read through Platter's mobile application. As I am not connected to Platter's application API I am simulating this data stream by reading into Kafka producer from a .csv file. I inject random time delays to simulate real life scenario and then write onto a topic "scanned_invoice". Spark Streaming act as a consumer and read the records from Kafka topic. When scanned invoice is read by Spark Streaming consumer, I check for restaurant_id and item_id in the incoming record. Based on this key (restaurant_id, item_id) record is fetched from Lookup table. Price for fetched item is now compared with price of new incoming item difference and % change in the prices is calculated. As I do not have access to the mobile API so I store the results to Feedback table and display the results on dashboard.

4.5 Serving layer

Serving layer serve user queries. Section 4.2 listed some common queries a user might look from serving layer. All these queries can be answered either directly from the views created by batch layer or speed layer or by combination of results from batch layer and serving layer. This layer not only cut the time for querying and generating views but also handle several concurrent views on the data. All the views are stored on Cassandra and the users can query these views from dashboard. Data was presented using Flask, a micro web framework written

Figure 4.8: *Speed layer pipeline*

in python. The application was deployed on Tornado, scalable, non-blocking web server that allow simultaneous queries on the serving layer. Tornado can scale to tens of thousands

of open connections making it ideal for web sockets and applications requiring long-lived connection to each user. Visualizations have been done using Highcharts, framework for making interactive charts on web pages.

4.6 Data organization

Data has been organized in this project on Cassandra. Before understanding data organization let's understand reasons for choosing Cassandra for this project. Data provided by Platter is structured and contains purchase transactions of a restaurant. These transactions occurred over a period of time and therefore represent time series. Cassandra is a key-value store and choice of a column used as primary key can significantly impact the performance of queries. Views generated for one restaurant are independent of views generated for another restaurant. Therefore, restaurant_id can serve as primary key. Cassandra is a distributed database where data is split across different partitions. Therefore, restaurant_id can serve as partition key. As Cassandra offer flexible schema it gives independence to integrate new data sources and add more features in future. Therefore, Cassandra is a perfect choice for this project. Master dataset is stored in MasterData table. New data is added to this table.

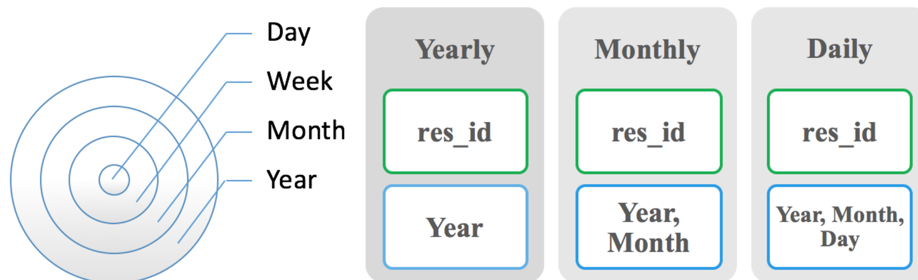

Figure 4.9: *Different views for time based queries.*

Views generated by serving layer for different time periods are stored on different tables - YearlyView, MonthlyView, WeeklyView, DailyView. The partition key remains same for all tables except for the clustering keys. For yearly view clustering key is Year and item_id whereas for a monthly view year, month and item_id act as partition key. Batch layer also

generate a "Lookup" table that contain latest information on each unique item purchased by a restaurant. This information includes various statistics about an item such as mean price, standard deviation, % change in prices, etc. The Lookup table will be used by speed layer to compare the price of incoming items scanned by the user. Any variations in price will be calculated using the last stored price for the item that was paid by the user. Data from speed layer is added to the master dataset and the results of comparison are stored in Feedback table. This table contain data for recently scanned invoices. Following diagram visualize the schema of Cassandra table.

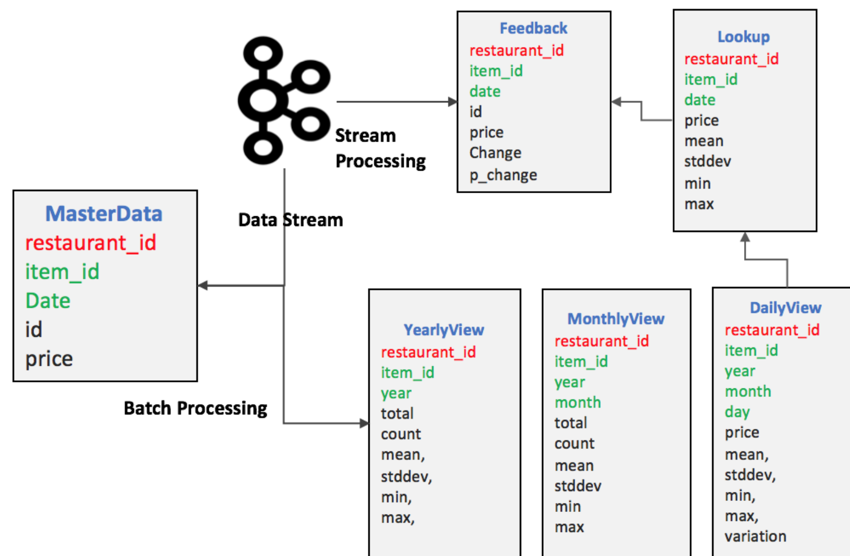

Figure 4.10: *Cassandra schema*

I created a keyspace Platter, with following tables to store view for serving layer:

1. **MasterData** - restaurant_id, item_id, date, price, id
2. **YearlyView** - restaurant_id, item_id, year, price, total, count, mean, stddev, min, max
3. **MonthlyView** - restaurant_id, item_id, year, month, price, total, count, mean, stddev, min, max
4. **DailyView** - restaurant_id, item_id, year, month, day, price, total, count, mean, stddev, min, max

5. **Lookup** - restaurant_id, item_id, date, price, mean, stddev, min, max

6. **Feedback** - restaurant_id, item_id, date, price, change, p_change,count

4.7 Infrastructure

Complete infrastructure was deployed on Amazon Web Services Elastic Compute Cloud (EC2), a web service providing secure, flexible compute capacity in Amazon cloud platform. Following diagram gives an overview of the infrastructure and tools that were used for the project.

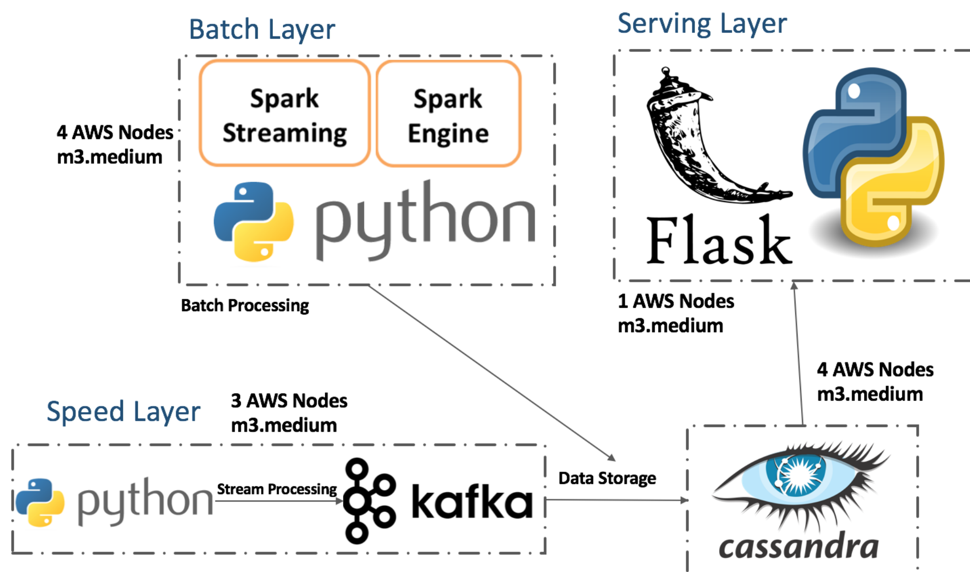

Figure 4.11: *AWS infrastructure*

- **cluster:** 12 X m3.medium
- **processor:** 1 CPU and 3.75 GiB memory
- **storage:** SSD Storage
- **cost/node:** $0.067/hour
- **cluster cost:** $578.88/month and $6946.56/year

Chapter 5

Results

I built a web application where users can login to their account and see different views pre-populated on their dashboard.

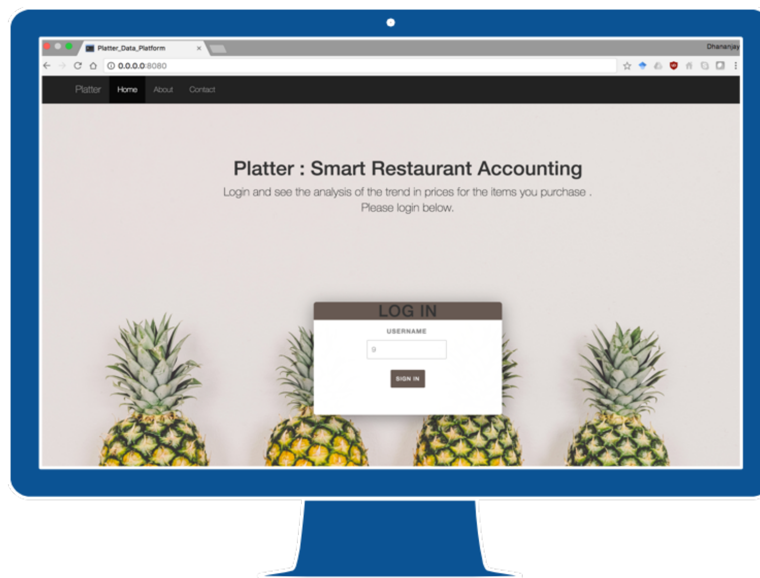

Figure 5.1: *User login*

Views displayed for user on its dashboard is generated by the serving layer using views that were generated by the batch layer and the speed layer. Batch layer was run on the master dataset which had nearly a million records. Batch data was stored on S3 and loaded to HDFS. The data is then processed on spark a for nearly an hour and results are stored onto various Cassandra tables. For example, the view for top 5 items by price during year 2016

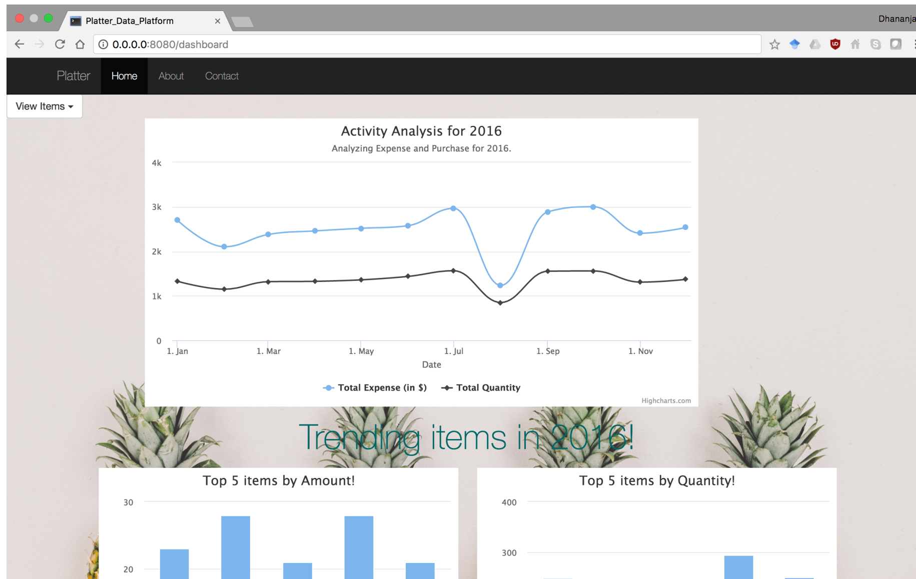

Figure 5.2: *Trend analysis for previous year*

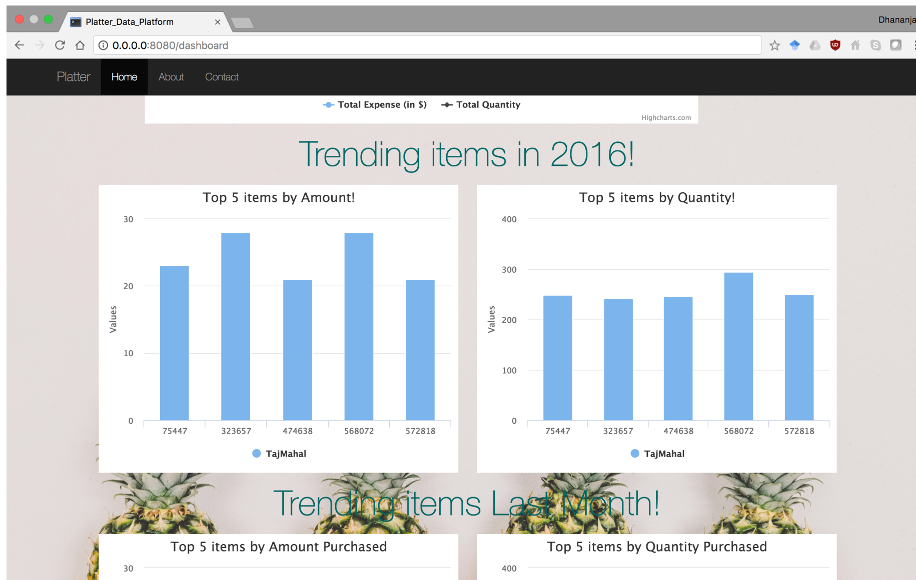

Figure 5.3: *Top 5 items during previous year*

was generated using YearlyView table. The speed layer process data received by scanning invoices. I tested this layer for reading nearly 100 records every second. Whenever a record is scanned and pushed on Kafka topic by producer it picked by spark streaming and compared to last price for the item on lookup table and variations are computed and stored on Feedback table. If a user is interested to get details about specific item, user lookup the item and the details are pulled by serving layer from the Lookup table.

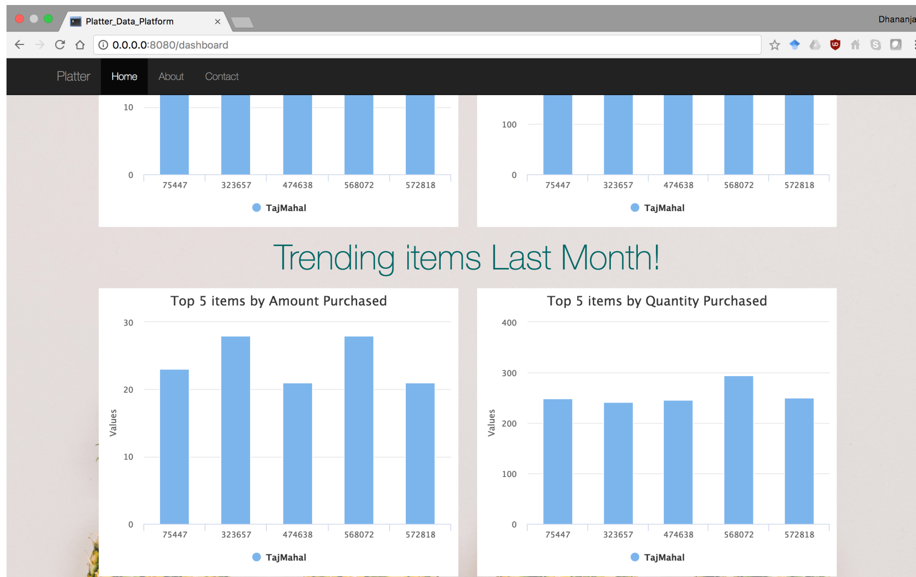

Figure 5.4: *Top 5 items from last month*

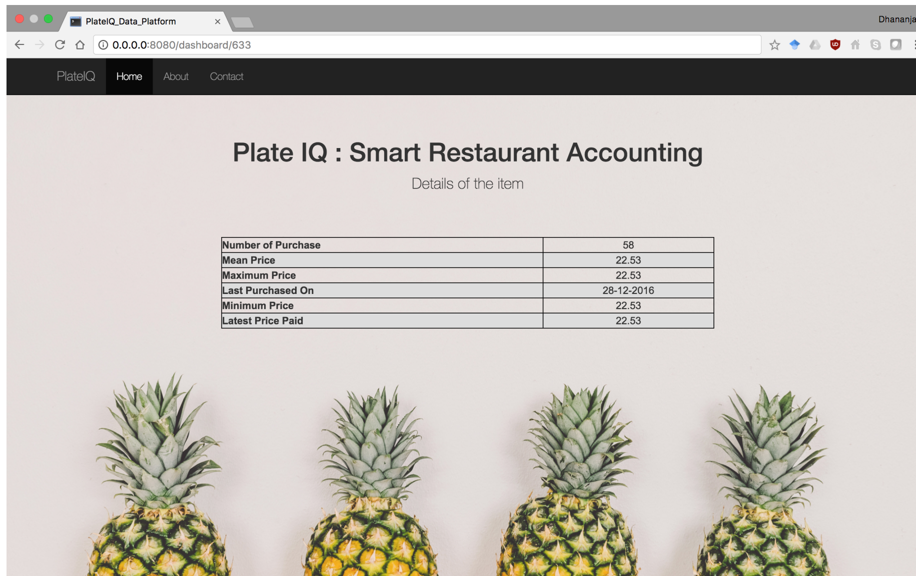

Figure 5.5: *Details of individual item*

5.1 Performance improvement

There is a huge gain in performance with new architecture. These gains were achieved mostly through queries from users. In the existing design each query was computed against SQL table for each item or time frame. This would need various joins and sub-queries. Also, updates are performed on data. All these operations make existing design slow and inefficient. With new design no update, joins or sub-queries are needed. This give a huge gain in terms of computation as all the queries have O(1) time complexity.

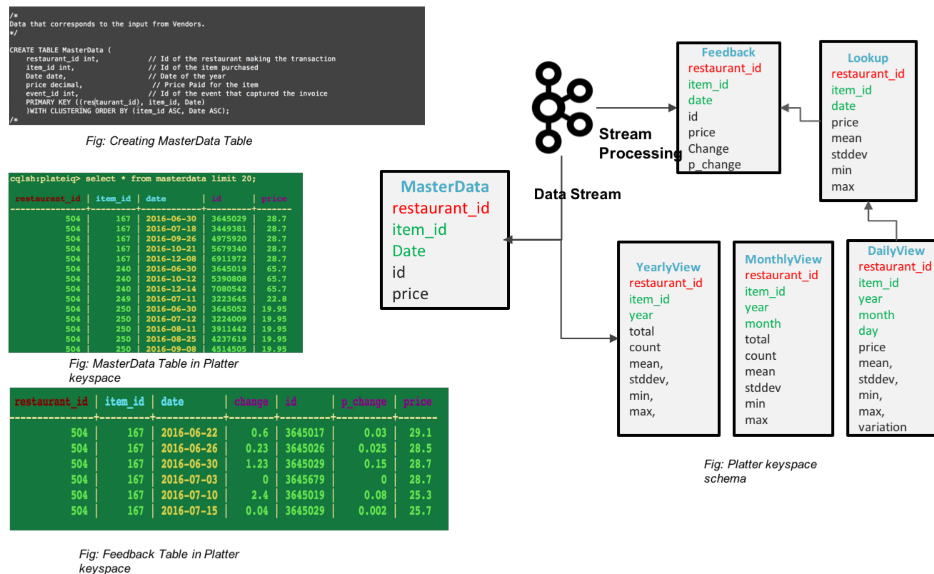

Figure 5.6: *Data organization for Platter*

Results of the new architecture can be summarized below:

1. Zero latency in generating query results due to precomputed views.
2. User dashboard is pre-populated with basic views with zero latency leading to a better user experience.
3. Common queries for example top 5 items for expenditure are generated.
4. Data from application is processed in real-time.
5. Platform is scalable to include additional data sources in future.

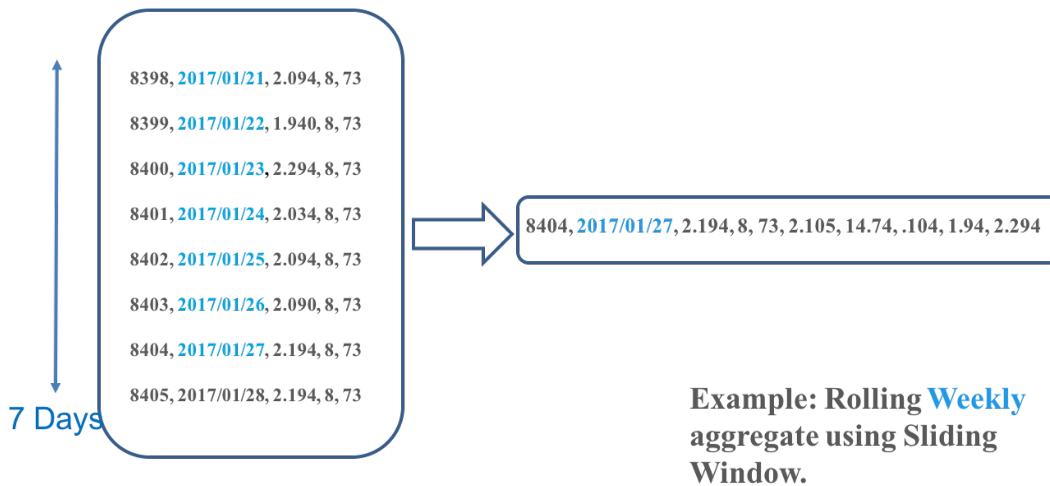

Figure 5.7: *Visualizing query: weekly aggregate-1*

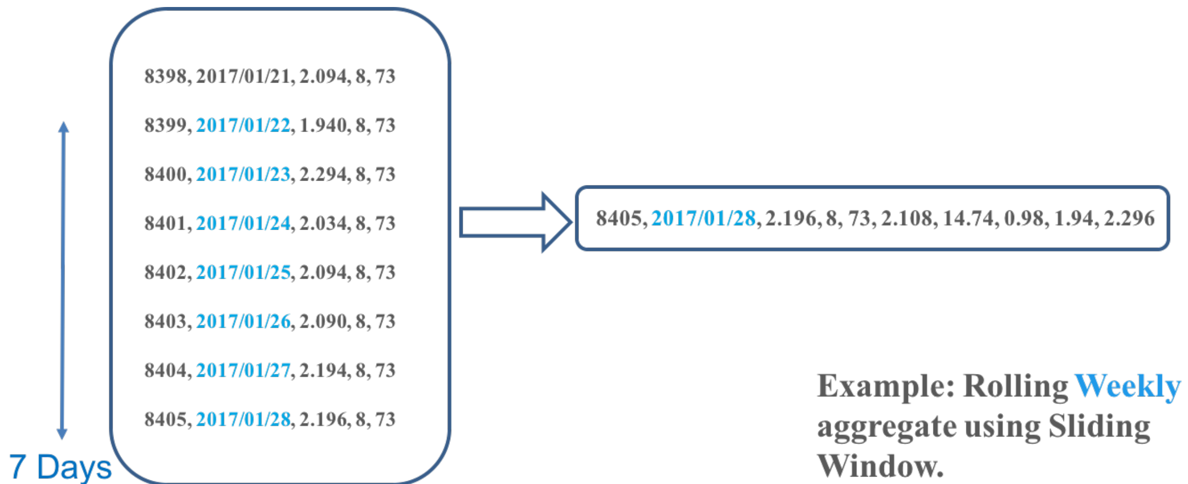

Figure 5.8: *Visualizing query: weekly aggregate-2*

5.7 and 5.8 visualize a sample query to calculate weekly average. This average is being calculated for last 7 days on a sliding window fashion. Window is a function on spark and the size can be changed based on user requirements. Different functions such as mean(), min(), max(), stddev() etc. are provided in pyspark.sql.function() library. 5.9 visualize query to calculate top K items during a year. In the example I represent calculating top K items by price paid during 2016.

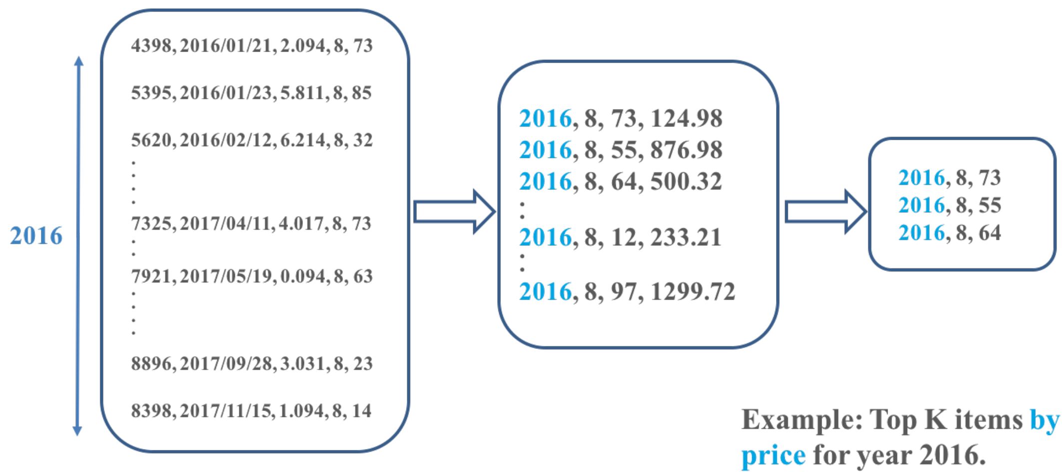

Figure 5.9: *Visualizing query: top-k items*

5.2 Testing

Testing was a challenge for the project as the dataset was small and limited. Most of the data was simulated and was not enough for stress testing. Also, as data is received from only one source system can not be tested for flexibility or capability to handle new data forms.

Chapter 6

Challenges

The objective of this project was to build a distributed data platform which is more efficient, scalable and fault tolerant than Platter's existing data platform. As Platter gains popularity, its user base will increase and so will the usage of its application. Although the platform incorporates the principles of a data supply chain and used Lambda Architecture (LA) to for building it, there were several challenges that I faced while building this system. These challenges have been illustrated below:

1. The first challenge was requirement gathering from Platter. It was important to understand features needed from this project and limitations of existing system. I organized stand-up calls with the team at Platter to understand and lay down the objectives of the project in the first phase.

2. Second challenge was understanding existing architecture at Platter. This was important to understand shortcomings and missing features from their platform. The stand-up calls in early stages helped to solve this problem. Platter currently deploy their infrastructure on Heroku, uses RabbitMQ and Celery for message queue, and Postgres for storage. I needed to decide if my project would be independent, or coupled with existing model. I eventually developed my project on Amazon EC2 with no connection to existing infrastructure. This not only removed any dependency but also ensure that the application can be tested without changes to any other components.

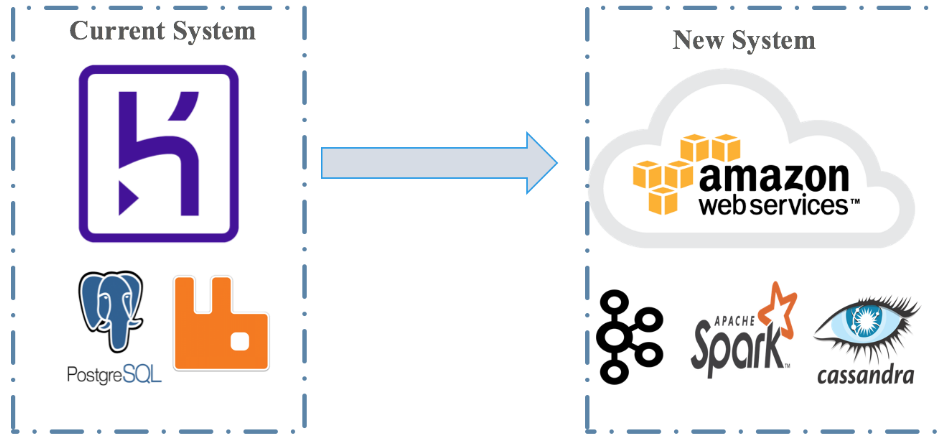

Figure 6.1: *Existing vs new architecture*

3. The most important challenge was choosing right tools to build the platform. As highlighted in chapter 1, there are several big data tools available. This make it important to choose right tools to serve the needs and requirements of the platform to be built. I read several white papers and case studies based on various tools to decide what suit best for the project. These tools suited to various needs of the project. For example, Spark was used over Hadoop because Spark not only offers better speed and efficiency but also because data was structured and SparkSQL library was easier to develop the code. Cassandra was used for data organization because it was scalable, cheap and data is perfect to be organized and query effectively. Kafka was chosen over RabbitMQ because Kafka offers a high throughput for write and offer a pubsub model allowing user to add different data sources. Therefore, choosing the right tools to build this project was very important.

4. Another challenge was to make the platform cost effective. Therefore, along with stability, fault tolerance and scalability cost was important factor. Therefore, I chose to use open source tools and technologies for this project. Let's consider a simple example of a trade-off that was made due to cost. Before deciding to use Cassandra for data organization I looked up for Redshift and InfluxDB but the cost of maintaining a Redshift cluster is nearly ten times more than maintaining a Cassandra cluster of

similar size. Hence I decided to use Cassandra.

5. As mentioned in section 3.4, data organization is important to ensure better read and write performance. It was a challenge to chose data organization platform and design the schema to organize data. I read several articles and literature on Cassandra and its use cases. I understood requirements of the project and how Cassandra fit into this project based on the nature of data and queries users might request from database. Also, I compared it with other databases such as Amazon Redshift, or InfluxDB could fit into this application.

6. Testing was a major hurdle for the project due to limited amount of data. The data was simulated to test the project at scale. The system was built on simulated data but neither batch layer nor speed layer could undergo a stress test to see if system would fail at certain point. Stress testing is definitely a future extension to this project. Also, as data comes from a single source and is organized it limits the ability to test the platform for handling different data sources.

7. Platter's in-built OCR on mobile application has a 2% error rate which means there are 2 receipts with error for every 100 receipts scanned. Currently a faulty reading is handled manually but as users increase this volume will be large. As data is fed into master dataset any incorrect reading would corrupt the data. Also, as an instant feedback is provided to user for scanned items incorrect reading would lead faulty feedback. To handle this if compared new price with the last paid price on Lookup table. If the price is more than five times I remove the record.

8. Another major challenge was data. As mentioned earlier, the dataset provided was small and most of the data was simulated. Limited data source and limited features in data limited possibilities for implementing additional features to use machine learning. This can be certainly be next step.

Chapter 7

Conclusion and future work

7.1 Future work

This project can be considered as most viable product for Platter. There are several additional features that can be extended to this MVP. I have listed these possible extensions below:

1. *Richer dataset:* The data provided by Platter was not rich in features that make it unsuitable for advance analytics. It is possible to create a feature rich dataset by combining different data sources such as market price data, location data, weather data, demographics data etc. Adding these data sources would help get better predictions for price and requirements.

2. *Using machine learning models:* This project calculates basic stats on data such as mean, std dev, min/max. One vital extension to the project can be ability to use machine learning algorithms to gain better insights. For example, predicting prices of a food items during a season using time-series models such as Hidden Markov model(HMM); predicting average consumption of food items using Logistic Regression. Such insights will be helpful in several ways such as stocking food supplies based on availability or consumption, dynamic pricing based on the predictions, change recipe ingredients based on the availability of items. Once a feature rich dataset is created

these advanced analytics can be easily performed.

3. *Adding resource scheduler such as Airflow or Luigi:* Users are presented with views for different time periods - year, month, day. When deployed to production jobs will run across different times of a year. Certain jobs are dependent on other jobs, for example Lookup table is created after jobs calculating DailyView is finished. In order to avoid deadlock and bottleneck in the jobs a resource scheduler is a perfect solution. Resource scheduler such as Airflow create a dependency DAG and help to avoid such problems.

4. *Using in-memory database:* In memory databases such as Redis would be a great fit for Feedback table which holds the latest items that were purchased by the restaurant. Such databases will also be very helpful to cache most commonly search items by a user and help reduce time to serve the queries.

7.2 Conclusion

Big data has made a significant impact on organizations, big or small across all domains of industry. Big data will continue to drive the growth of organizations, data volume will continue to grow, and there will be more tools to work with. All organizations will embrace this technology eventually. But, the most important challenge will be designing a big data platform to answer following questions. How to get 'relevant' data source systems? How to store 'relevant' data? How to organize data in most efficient way to ensure fast readability? How to process real-time data along with batch data? How to ensure low-latency to users? Big data platform is not a data munching machine but it needs to be designed meticulously. This project proposed 'data supply chain management' that laid principles of designing a big data platform. Lambda Architecture provide an framework to enact these principles to builds a scalable and fault-tolerant data processing platform for real-time processing with low latency. This project was implemented using LA based on the principles of data supply chain for Platter. The principles laid down in this project can be extended to any organization whether big or small to build a low cost, scalable and fault-tolerant data platform.

Bibliography

[1] Bernard Marr. Big data: 20 mind-boggling facts everyone must read, 2015. URL https://www.forbes.com/sites/bernardmarr/2015/09/30/big-data-20-mind-boggling-facts-everyone-must-read.

[2] Sanjay Ghemawat Jeffrey Dean. Mapreduce: Simplified data processing on large clusters, 2005. URL https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf.

[3] Nathan Marz and James Warren. *Big Data: Principles and Best practices of scalable real-time data systems*. Manning Publications Co., Shelter Island, NY, 2015.

[4] Wikipedia. Flow of data through the processing and serving layers of a generic lambda architecture, 2014. URL https://en.wikipedia.org/wiki/Lambda_architecture.

[5] KnowledgeBrief. Supply chain management, 2016. URL https://www.kbmanage.com/concept/supply-chain-management.

[6] Accenture. Building a modern data supply chain to accelerate data, 2016. URL https://www.accenture.com/us-en/insight-data-acceleration-modern-data-supply-chain.

[7] Apache Software Foundation. The apache software foundation blog, 2016. URL https://blogs.apache.org/foundation/date/20160217.

[8] Nathan Marz. How to beat the cap theorem, 2011. URL http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html.

[9] Apache Hadoop. Hdfs architecture guide, 2008. URL https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[10] Apache Spark. Spark sql, dataframes and datasets guide, 2016. URL http://spark.apache.org/docs/latest/sql-programming-guide.html.

[11] Apache Spark. Apache spark library, 2016. URL http://spark.apache.org/streaming/.

[12] Jacek Laskowski. Spark architecture, 2015. URL https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-architecture.html.

[13] Apache Kafka. Apache kafka documentation, 2016. URL https://kafka.apache.org/0100/documentation/.

[14] Apache Kafka. Apache kafka topics and logs, 2016. URL https://kafka.apache.org/intro.html.

[15] Robert Greiner. Cap theorem: Revisited, 2014. URL http://robertgreiner.com/2014/08/cap-theorem-revisited/.

[16] Benjamin Erb. *Concurrent programming for scalable web architectures*. Open Access Repositorium der Universitt Ulm., universitt ulm, 2012.

[17] Ebay Jay Patel. Cassandra data modeling best practices, part 1, 2012. URL http://www.ebaytechblog.com/2012/07/16/cassandra-data-modeling-best-practices-part-1/.