Prototyping a Data Analysis Capability Leveraging Lambda Architecture Concepts

Donald A. Van Patten, Douglas D. Hodson, Michael R. Grimalia, and Ryan D. Engle
Department of Electrical and Computer Engineering
Air Force Institute of Technology
Wright-Patterson AFB, Ohio 45433
Email: {donald.vanpatten, douglas.hodson, michael.grimalia, ryan.engle}@afit.edu

Abstract—From 2013 to 2015, more data was created than in the entire history of humanity up to that point. In 2013, the amount of data in the known universe was estimated at 4.4 zettabytes (ZB) and was on pace to double in size every two years. If this growth continues, by 2020 there will be roughly 40 ZB, or 40 billion terabytes (TB), of data in existence. This paper presents an integrated solution to display and analyze data using modern web technologies with a traditional relational database. Our latest prototype architecture embraced Lambda architecture concepts to enhance its capabilities.

I. INTRODUCTION

From 2013 to 2015, more data was created than in the entire history of humanity up to that point [14]. In 2013, the amount of data in the known universe was estimated at 4.4 ZB and was on pace to double in size every two years [10]. If this growth continues, by 2020 there will be roughly 40 ZB, or 40 billion TB, of data in existence [10], [9]. This data is generated by many different sources: YouTube reports its users upload 400 hours of video every minute [12]; Facebook’s 1.18 billion daily active users [4] upload more than 350 million new photos every day, adding to an estimated more than 700 billion photos already stored by Facebook [18]; the New York Stock Exchange generates more than 4 TB of data each day [8]; Dropbox users upload more than 800,000 new files every minute to the cloud storage service [12]; and the English-language Wikipedia which contains upwards of 5.2 million articles, grows at a rate of 20,000 articles per month, and has roughly 1 gigabyte (GB) of compressed text added per year [23].

Datasets of this size contain a lot of information, but that information does little good for an organizational owner if it cannot be accessed or analyzed efficiently. Traditional ways of storing and managing data, such as relational database management systems (RDBMSs), are quickly overwhelmed by data in this quantity [17]. Companies like Google, Facebook, and Amazon must deal with situations like this, where large, complex datasets as part of their daily business. Google queries over 130 trillion web site indices totaling over 100 billion GB [7] to process the more than 4 million search queries it receives every minute [11]. Facebook processes over 30 petabytes (PB) of data [9] generated by its user base to serve advertisements, make friend recommendations, and display news articles tailored to a user’s interests. Add server log messages, blog posts, news articles, sensors from mobile phones, and wireless sensor networks and its easy to understand how 30,000 GB of data are created every second [17].

These companies, and many like them, rely on the paradigm of “big data” to facilitate relatively quick analysis of large, complex data sets in order to extract meaningful information. A term that has become as nebulous as “the cloud,” big data is used to describe extremely large datasets which, when analyzed, can reveal trends and patterns and can “extract value from very large volumes of a wide variety of data.” [5]

This paper presents an architecture to support data analysis using modern web technologies and big data concepts.

II. BACKGROUND

We first provide background information on the technologies utilized to construct a web-based data analysis capability.

A. Angular 2

In 2014, Google announced a new version of the AngularJS framework called AngularJS 2.0, or more simply Angular 2 [6], [20]. Angular 2 is a rewrite of AngularJS, it is written in Microsoft’s TypeScript language (a superset of JavaScript) and focused on mobile-first web application development. Angular 2 foregoes any backward compatibility with older browsers in order to leverage modern “web technology in simpler and more direct ways.” [6]

Angular 2 relies on the Shadow Document Object Model (DOM) capabilities, which create “isolated DOM trees to prevent collisions such as duplicate identifiers, or accidental modifications” [20] and allows developers to create an encapsulated DOM subtree which contains “markup, Cascading Style Sheets (CSS), JavaScript, or any asset that can be included in a web page.” [20] This encapsulation provides a way to create isolated web components that won’t impact or interfere with others.

The building blocks of Angular 2 applications are Directives, Components, and Services [20]. Directives enable Angular 2 applications to define behaviors for DOM elements and Components extend Directives by attaching a view template in which HyperText Markup Language (HTML) code is contained. Directives and Components both consist of a custom element selector tag and an exportable TypeScript
class tag, but only Components contain an additional HTML template tag. This template tag encapsulates the HTML and provides the view for the associated Component. Finally, Services contain the “business logic” of an application for better “separation of concerns, maintainability, and reusability of code” [20] and to help adhere to the single responsibility principle [15].

One final major piece of the Angular 2 application concerns building single-page applications (SPAs). A SPA is an HTML document that displays content in a single page. Content is dynamically displayed to the screen without the need to refresh the entire page, thus creating a desktop application-like feel [20]. Angular 2 is able to build SPAs through the use of Components and the Angular 2 component-based router [6]. The router renders content defined in the template tag to the page whenever a user clicks a link to a new view (Component). The router provides the mechanism, and combines the other Angular 2 building blocks, to create a responsive web application that mimics the functionality, feel, and richness of a desktop application (i.e. thick clients).

B. Data-Driven Documents

Data-Driven Documents (D3) is an open-source JavaScript library that provides interactive data visualization for browser-based web application. D3 is one of many tools available to web developers which enables displaying data in the browser. D3 relies on existing web standards (e.g. HTML, Scalable Vector Graphics (SVG), CSS) and native browser technologies (i.e. no plug-ins required) to perform its visualizations [2], [3], [19]. This is unlike most other web data visualization frameworks and charting libraries which create their own standards or rely on plug-ins, like Java, to provide data visualization.

Additionally, D3 is not a charting library as it does not provide predefined visualizations [3], [19]. As such, any visualization must be defined completely by the developer. This usually results in a steeper learning curve when compared to other charting libraries – which abstract much of the work for the user [3], [19]. However, this increased complexity also provides much greater flexibility in providing web-based data visualizations as more customization is permitted with D3 than with other libraries [19].

D3 operates by loading provided data into the browser’s memory, binding those data to elements (HTML, SVG, CSS) through the use of selection statements, transforming the elements according to predefined settings (e.g. scale, color, sorting rules), and providing transitions when prompted by user input or interaction [19].

D3 binds data to elements and manipulates those elements through the use of selection statements. These selection statements utilize JavaScript and the DOM to select one or more HTML, SVG, or CSS elements. The elements can be selected by their associated tag, class selector, ID selector, or attribute value [2], [3], [19]. Once selected, D3 can manipulate those elements by modifying their attribute fields or applying CSS styles.

D3 provides interactivity to the displayed data by binding JavaScript event listeners to D3 selections (selected DOM nodes) [19]. These listeners trigger an event when the user interacts with the specified element and provide a robust, interactive data visualization experience.

C. Lambda Architecture

The Lambda Architecture was introduced by Nathan Marz in 2011 [16] as a way to more efficiently answer arbitrary questions on big data-sized datasets. It is built on the principle that making the data in the system immutable reduces system complexity, thereby reducing the possibility of accidentally introducing errors to the system and protecting the integrity of the data over the long term.

The Lambda Architecture defines a generalized framework that is robust, hardware and human fault-tolerant, linearly scalable, extensible, easily debuggable, and require minimal maintenance [17]. It defines an architecture or pattern to produce accurate results from algorithms operating on large datasets while still enabling near real-time access to the latest data in the system and keeping the complexity of the system low.

The Lambda Architecture derives its name from the fact that the input data stream splits into two separate streams (see Figure 1), resembling the Greek letter lambda (λ). These data streams are directed (or forked) and sent to two separate processing systems, or layers, each with a specific purpose: the batch layer, which processes the data and creates batch views for the serving layer, and the speed layer, which generates real-time views of the latest available data.

D3 provides interactivity to the displayed data by binding JavaScript event listeners to D3 selections (selected DOM nodes) [19]. These listeners trigger an event when the user interacts with the specified element and provide a robust, interactive data visualization experience.

C. Lambda Architecture

The Lambda Architecture was introduced by Nathan Marz in 2011 [16] as a way to more efficiently answer arbitrary questions on big data-sized datasets. It is built on the principle that making the data in the system immutable reduces system complexity, thereby reducing the possibility of accidentally introducing errors to the system and protecting the integrity of the data over the long term.

The Lambda Architecture defines a generalized framework that is robust, hardware and human fault-tolerant, linearly scalable, extensible, easily debuggable, and require minimal maintenance [17]. It defines an architecture or pattern to produce accurate results from algorithms operating on large datasets while still enabling near real-time access to the latest data in the system and keeping the complexity of the system low.

The Lambda Architecture derives its name from the fact that the input data stream splits into two separate streams (see Figure 1), resembling the Greek letter lambda (λ). These data streams are directed (or forked) and sent to two separate processing systems, or layers, each with a specific purpose: the batch layer, which processes the data and creates batch views for the serving layer, and the speed layer, which generates real-time views of the latest available data.

![Fig. 1. Traditional Lambda Architecture flowchart](image-url)
1) The Master Dataset: The master dataset is considered the “source of truth” [17] and is at the core of the Lambda Architecture. In contrast, in traditional RDBMSs which stores mutable data that is updated by overwriting records – the current state of the system is kept. This approach limits the questions that can be asked, because historical data is lost forever when it is overwritten. Conversely, the Lambda Architecture opts to store the historical state of the system by constantly adding data to the system and, in the process, effectively removing an update functionality. This is accomplished by enforcing strict properties of data in the system, such as rawness, immutability, and perpetuity. These properties limit data corruption, improve system performance, and ensure data deemed valuable by the system owner is kept in the system to answer those questions a mutable schema RDBMSs cannot answer.

Rawness, in a data-science context, is information “held to be true simply because it exists” [17]. Raw data is data which cannot be derived from any other information. This provides a mechanism to maximize the insights which can be gleaned from the data, as opposed to storing summarized data which loses granularity [17]. For example, a birthdate is considered raw data, as an age can be computed from a birthdate, but the reverse is not true. The master dataset is configured so that data in the Lambda Architecture is stored in the rawest possible form.

Immutability is the second key concept of data in the master dataset. Immutability means that raw data is stored and never changed via “update”. This eliminates the update and delete functions of the create, read, update, and delete (CRUD) operations normally implemented in traditional RDBMS applications. This defines a human fault-tolerant data model because, save a full system failure, no data can be lost. In the case that bad data is written to the master dataset, timestamps indicating “freshness” provide a mechanism to delete the errant data and recover back to the last known good state.

The final key concept of data in the Lambda Architecture is perpetuity. The perpetuity, or truthfulness, of the data in the Lambda Architecture is a byproduct of the immutability property, combined with timestamps. This combination provides the ability to determine when a fact in the database was true based on its timestamp. To illustrate this point, take the population of the United States. The 1910 Census concluded that the population of the United States was 101,115,487 [21]. In 2010, the United States Census Bureau determined the population to be 308,745,538 [22]. Because there is a specific date associated with both statistics, the data is eternally true. Without the date, there is no way to validate which number is the current number or which is true at all, calling into question the reliability of the data itself. Additionally, the fact that the population has increased (or decreased) in subsequent censuses is captured in additional data stored by the census bureau. This illustration captures the concept of perpetuity in the Lambda Architecture – when combined with timestamps, data that is true at a specific point in time will always be true.

2) The Batch Layer: The batch layer is responsible for processing and storing the master dataset. This layer maintains precomputed answers to questions often asked of the data (called batch views). In general, the batch layer takes advantage of distributed processing, allowing it to process large amounts of data efficiently. Because batch layer computations are performed using the entire dataset, and because the size of the dataset is often very large, generating batch views is a time consuming operation. Thus, batch views are computed on a frequent basis, any errors introduced into the system will be remedied the next time the batch operations are performed after the error is corrected. Second, processing all of the data in the dataset every time provides a level of accuracy that is hard to achieve with incremental algorithms [17]. After batch views are generated, they are sent to the serving layer where they are made available for querying by a user.

3) The Serving Layer: The serving layer indexes views from the batch layer and exposes them to the system. This provides low-latency batch layer results to queries. The serving layer is a distributed database which is specially chosen to provide random, read-only access to the views created by the batch layer. When a new batch operation completes, the view updates, providing access to the data in the master dataset at the time of the last batch operation. Because the new views are written into the serving layer database at one time, there is no need for a database which supports random write access. This greatly reduces the complexity of the system [17] and keeps with the Lambda Architecture mantra of less complexity equals greater fault tolerance.

4) The Speed Layer: The batch and serving layers work together to provide accurate results to frequently asked questions of the data using the entire dataset. There is a cost, however, associated with continuously reprocessing the entire dataset: high-latency – processing big data-sized datasets takes a long time. This means, depending on how often new data arrives to the system, that the latest computed views might not contain the latest data available to the system. The speed layer addresses this problem and ensures the newest information is available, in near real-time, to the system. The speed layer processes only that data which has recently arrived (the data not currently contained in a precomputed batch view). As soon as new data arrives, the speed layer runs the same algorithms used in the batch layer on only the new data to generate and/or update real-time views. When combined with the batch views, this provides an up-to-date view of the data and ensures any analysis done on the dataset includes the latest information.

This incremental approach stands in stark contrast to the computation approach of the batch layer. Incremental calculations bring with them complexity, because it is necessary to keep track of the previous results in order to compute results for the new data. However, the complexity is isolated to the layer where the size of the data in the smallest and where
the results are only temporarily stored. Therefore, any errors introduced, whether algorithm errors or approximation errors, will eventually be expunged from the speed layer once the data makes its way through the batch and serving layers.

III. EXISTING ARCHITECTURE

An existing web architecture was created using multiple frameworks; one for the client-side and another for the server-side applications. The server-side utilizes Node.js to serve client-side code to the user (i.e., browser) and acts as the interface to the relational database. The client-side application uses Angular 2 to provide SPA capabilities and interface with the server-side application, which retrieves data from a relational database to display.

A. Server-Side Architecture

The server-side architecture, written as a Node.js-based JavaScript application, provides a number of capabilities. First, it serves the client-side Angular 2 code when requested by the browser. Secondly, it provides the communication sockets between the browser and the database hosted on Amazon Web Services (AWS). Finally, when a user initiates a request for data, the Node.js server sends the appropriate Structured Query Language (SQL) query to the relational database and forwards the response to the browser.

Upon serving the Angular 2 application to the user, the server opens a socket and listens for a connection request from the client application utilizing the Socket.IO JavaScript library. Once a successful connection is established, the client is able to pass information to, and receive information from, the server. When the user clicks on a button, for example, “Draw Chart”, the Angular 2 application submits a request for data to the Node.js server via the open socket. Upon receiving the request, the Node.js server then sends a pre-defined SQL request to the database for the appropriate data and waits for a response. Once the response is received, the data is forwarded back to the browser through the open socket and is displayed by the Angular 2 application.

B. Client-Side Architecture

The Angular 2 SPA is a prototype, proof-of-concept to provide data analysis functionality using web technologies. It utilizes the D3 JavaScript library to build interactive graphs of the data provided by the server. It interacts with the relational database hosted on AWS via the Node.js server discussed in Section III-A.

When the client’s browser receives the Angular 2 application, it attempts to establish the socket connection mentioned in Section III-A. Angular 2 loads the SPA and a user interface is drawn. It provides the ability to draw a chart using server stored data. Once the user requests the data and the server returns the requested information, the data is parsed, loaded into D3 and displayed on the screen in the form of an interactive graph.

IV. LAMBDA ARCHITECTURE PROPOSAL

The Lambda Architecture, being a general framework, is tool agnostic and there are many big data tools available that fit into the Lambda Architecture framework; therefore, picking the right ones for a specific application is no easy task. One of the major challenges is that different tools developed by different developers require the business logic to be replicated and synchronized in multiple places and in multiple different languages [13]. This makes picking the right tools even more important as it has an impact on the complexity of the overall design. Fortunately, a few tools exist that reduce this complexity and simplify the process of synchronizing and maintaining the algorithms used to create the Lambda Architecture solution proposed here.

A. Batch Layer

One such tool is Apache Spark, which provides a unified ecosystem for performing both batch and streaming data operations using the same code base. Spark Core, Spark SQL, and Spark Streaming are three components of Spark that prove useful in the context of data analysis.

In the batch layer, Spark Core provides batch processing of the master dataset. Functions performed on the master dataset to generate batch views can be written in Java, Scala, Python, or R [1]. This is enabled by the Spark application program interface (API) which provides language-specific hooks into Spark and its modules.

Spark SQL is one such module which provides access to SQL databases and structured and semi-structured data in the batch layer. Through the use of a MySQL Java Database Connectivity (JDBS) connector, Spark can connect to the RDBMS that stores the data of interest. This provides a way to ingest the data from either the current database solution or a big data solution like Hadoop and the Hadoop Distributed File System (HDFS). An added benefit to using Apache Spark is that it performs equally as well with GB-sized datasets as it does when scaled to handle PB-sized datasets [24]. This allows the architecture to easily scale as the amount of data increases.

B. Serving Layer

Among the available choices for the serving layer, Cassandra is the best in this construct for handling data batch view outputs from Spark’s batch functions. Cassandra is an open-source, distributed NoSQL database management system (DBMS). It scales linearly, automatically distributes data across all nodes, and has built-in replication. The reasons for using Cassandra are multi-faceted. It is able to handle smaller databases (in the GB and TB range) along with larger, big data sets. It is easier to setup than other alternatives, such as ElephantDB or HBase, due to its masterless architecture where every node has the same role. It has a Spark connector, allowing it to interface with the batch and speed layer implementations of Spark. Finally, the Cassandra Query Language (CQL) used to query the database has syntax very similar to SQL providing a familiarity to anyone with SQL experience.
C. Speed Layer

For the speed layer of this Lambda Architecture implementation, Spark Streaming provides micro-batch processing of real-time, streaming data. Because Spark Streaming is a module built on the Spark Core, just as with Spark SQL, the functions written in the batch layer can be reused in the speed layer within Spark Streaming. The final piece in the Lambda Architecture is making the real-time views available to be queried by the system. This can be done by using the Cassandra DBMS in the serving layer to serve the real-time views from the speed layer. Batch and real-time views can then be easily joined when queries are received providing access to both historical and real-time information.

V. ANALYSIS OF LAMBDA ARCHITECTURE PROPOSAL

The Lambda Architecture solution proposed in the previous sections defines a big data framework for data to be ingested and processed. The proposed Lambda Architecture solution will limit the code base complexity by utilizing the same framework in the batch and streaming layers. This homogeneous code base, provided by the Spark framework, eliminates the need to synchronize batch and speed layer algorithms across different languages. This section analyzes the proposed Lambda Architecture solution.

Additionally, the proposed architecture utilizes Cassandra and its Spark connector to provide serving layer capabilities for both the batch and speed layers. Example implementations of the Lambda Architecture found during literature review varied in their implementation of the serving layer. Some utilized separate DBMSs for each, and some combined them. In the case of this proposal, Cassandra can be configured to meet the requirements for both the batch layer serving (batch writes, random reads) and the speed layer (random writes, random reads). Therefore, it can be used to serve results from both layers leading to a Lambda Architecture conceptual design shown in Figure 2. Figure 3 shows how each of the tools presented in the previous section coalesce into the framework.

Finally, Figure 4 shows the proposed Lambda Architecture solution as implemented in the overall data analysis system developed through the course of this work.

VI. FUTURE WORK

All of the processing necessary to parse and display the data is performed by the browser. When the data becomes large, this significantly slows the application and results in a large amount of resources consumed. Moving this processing from the browser to the Lambda Architecture to provide some or all of the pre-processing by presenting views about the data at different levels of detail would relieve the processing burden of the browser. In turn, this would speed up the application and provide a mechanism to control the level of detail presented.
to the user based on the user’s interaction with the data (e.g. displaying less data points when the user is zoomed out and progressively displaying more and more detail as the user zooms in).

VII. CONCLUSIONS
This work proposed an integrated solution to analyze data using modern web technologies and Lambda Architecture concepts. The Angular 2 application utilizes D3 to provide data visualization in the browser. The Lambda Architecture proposal integrates Apache Spark and Spark Streaming along with Cassandra to provide a big data database solution to process the data. Together, this system represents a solution to resolving a big data problem in relation to analysis of it. We believe this solution has applicability many problem spaces where large amounts of streaming data need to be analyzed visually.

DISCLAIMER
The views expressed in this paper are those of the authors and do not reflect the official policy or position of the United States Air Force, the Department of Defense, or the U.S. Government.

REFERENCES