Spring 1-1-2016

Engineering Scalable Distributed Services for Real-Time Big Data Analytics

Sahar Hussein Jambi

University of Colorado at Boulder, sahar.jambi@colorado.edu

Follow this and additional works at: https://scholar.colorado.edu/csci_gradetds
Part of the Engineering Commons, and the Theory and Algorithms Commons

Recommended Citation
https://scholar.colorado.edu/csci_gradetds/134

This Dissertation is brought to you for free and open access by Computer Science at CU Scholar. It has been accepted for inclusion in Computer Science Graduate Theses & Dissertations by an authorized administrator of CU Scholar. For more information, please contact cuscholaradmin@colorado.edu.
Engineering Scalable Distributed Services for Real-Time Big Data Analytics

by

Sahar Hussain Jambi

B.S., King Abdulaziz University, 1996
M.S., King Abdulaziz University, 2008
M.S., University of Colorado Boulder, 2013

A thesis submitted to the
Faculty of the Graduate School of the
University of Colorado in partial fulfillment
of the requirement for the degree of
Doctor of Philosophy
Department of Computer Science
2016
This thesis entitled:
Engineering Scalable Distributed Services for Real-Time Big Data Analytics
written by Sahar Hussain Jambi
has been approved for the Department of Computer Science

(Kenneth M. Anderson)

(Richard Han)

(Tom Yeh)

(Kai Larsen)

Date ________________________

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.
ABSTRACT

Sahar Hussain Jambi (Ph.D., Computer Science)

Engineering Scalable Distributed Services for Real-Time Big Data Analytics

Thesis directed by Professor Kenneth M. Anderson

There is high demand for techniques and tools to process and analyze large sets of streaming data in both industrial and academic settings. While existing work in this area has focused on a wide range of issues including persistence technologies, advanced analysis tools, functional web interfaces, and the like, I focus on query support. In particular, I focus on providing analysts flexibility with respect to the types of queries they can make on large data sets, in real time as well as over historical data. I am building a lightweight service-based framework—EPIC Real-Time—that manages a set of queries that can be applied to user-initiated data analysis events (such as studying tweets generated during a disaster). My prototype combines stream processing and batch processing techniques inspired by the approach embodied in the Lambda Architecture. I investigate a core set of query types that can answer the wide range of queries asked by analysts who study crisis events. For this research, I design and develop a flexible set of real-time analytical tools that will allow analysts to ask new types of questions as they move their research activity from after a crisis to analysis during an event. This will enable them to monitor online social behaviors and capture interesting interactions in real-time across the various phases of a disaster. In this dissertation, I present a prototype implementation of EPIC Real-Time which makes use of message-driven and reactive programming techniques. I also present a performance evaluation on how efficiently the real-time and batch-oriented queries perform, how well these queries meet the needs of Project EPIC analysts, and provide insight into how EPIC Real-Time performs along a number of non-functional requirements important for big data, such as performance, usability, scalability, and reliability.
ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Professor Ken Anderson for his valuable guidance and direction. This thesis would not have been possible without his big support, kindness, and patience.

I would also like to thank Professor Leysia Palen for all her kind support and for including me as a member of Project EPIC.

Thanks to all of my software engineering colleagues—Mazin, Mario, Ali, Ahmet, Reem, Rsha, and Afnan, for their help and support.

Thank you to all of my colleagues in Project EPIC—Jennings, Marina, Melissa, Robert, Joanne, and Lise, who participated in many sessions of my interviews, for helping me learn about their research practices in crisis informatics research. Their cooperation was a great help in fulfilling my research goals.

Big thanks go to my home university, King Abdulaziz University, Jeddah, Saudi Arabia, that granted me a scholarship to get this degree, which I’m so grateful for.

Finally, thanks to my small family, Ehab and Joanna, for supporting me with the time, love and care I needed, and for my great mother and brothers, parents-in-law and sister-in-law Rehab, for their prayers and thoughts.
CONTENTS

CHAPTER 1 ................................................................................................................................. 1
INTRODUCTION ......................................................................................................................... 1

CHAPTER 2 ................................................................................................................................. 4
BACKGROUND ............................................................................................................................ 4
  2.1 Crisis Informatics Research ............................................................................................... 4
    2.1.1 Project EPIC: Current State .................................................................................... 4
    2.1.2 Project EPIC: Next Generation .............................................................................. 7
  2.2 Big Data Analytics ............................................................................................................. 8
    2.2.1 Big Data: An Emerging Paradigm .......................................................................... 8
    2.2.2 Big Data Paradigms: Batch Vs. Streaming .............................................................. 10
    2.2.3 Big Data System Architecture .............................................................................. 11
    2.2.4 Data Analytics: Common Methods and Types ....................................................... 13
    2.2.5 Stream Processing ................................................................................................. 15
    2.2.6 The Lambda Architecture ..................................................................................... 18
    2.2.7 Apache Spark ......................................................................................................... 20
  2.3 Modern Software Development Trends ........................................................................... 21
    2.3.1 Microservices .......................................................................................................... 23
    2.3.2 API Management .................................................................................................... 25
    2.3.3 Reactive Programming ......................................................................................... 25
    2.3.4 The Actor Model ................................................................................................... 27

CHAPTER 3 ................................................................................................................................ 29
ANALYSIS AND DESIGN .......................................................................................................... 29
  3.1 Research Questions ........................................................................................................... 29
  3.2 Investigative User Study ................................................................................................... 30
    3.2.1 Individual Interviews ............................................................................................. 31
    3.2.2 General Findings .................................................................................................... 45
  3.3 User Requirements .......................................................................................................... 49
  3.4 Types of Queries .............................................................................................................. 50
    3.4.1 Dataset Query ......................................................................................................... 50
    3.4.2 Trend Query ........................................................................................................... 51
    3.4.3 Popular Query ........................................................................................................ 52
TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1</td>
<td>INTERVIEW QUESTIONS</td>
<td>31</td>
</tr>
<tr>
<td>Table 2</td>
<td>INTERVIEWS RESULTS</td>
<td>47</td>
</tr>
<tr>
<td>Table 3</td>
<td>CASSANDRA TABLE SCHEMA - KEYSPACE AND MAIN TABLES</td>
<td>69</td>
</tr>
<tr>
<td>Table 4</td>
<td>CASSANDRA TABLE SCHEMA - QUERY META DATA TABLES</td>
<td>70</td>
</tr>
<tr>
<td>Table 5</td>
<td>CASSANDRA TABLE SCHEMA - QUERY VIEW TABLES</td>
<td>71</td>
</tr>
<tr>
<td>Table 6</td>
<td>SPARK JOB EXAMPLE OF SPEED LAYER</td>
<td>75</td>
</tr>
<tr>
<td>Table 7</td>
<td>SPARK JOB EXAMPLE OF BATCH LAYER</td>
<td>76</td>
</tr>
<tr>
<td>Table 8</td>
<td>SPARK JOB EXAMPLE OF SERVING LAYER</td>
<td>77</td>
</tr>
<tr>
<td>Table 9</td>
<td>EVENTS SERVICE API DESCRIPTION</td>
<td>81</td>
</tr>
<tr>
<td>Table 10</td>
<td>QUERY ENGINE API DESCRIPTION</td>
<td>82</td>
</tr>
<tr>
<td>Table 11</td>
<td>PERFORMANCE MEASUREMENT OF REAL-TIME QUERIES</td>
<td>95</td>
</tr>
<tr>
<td>Table 12</td>
<td>PERFORMANCE MEASUREMENT OF BATCH QUERIES</td>
<td>97</td>
</tr>
</tbody>
</table>
FIGURES

FIGURE 1: EPIC ANALYZE SOFTWARE ARCHITECTURE (ANDERSON, ET AL., 2015) .............................................6
FIGURE 2: BATCH VS. REAL-TIME ANALYTICS ........................................................................................................8
FIGURE 3: BIG DATA TECHNOLOGY MAP FOR THE BIG DATA VALUE CHAIN (HU, ET AL., 2014) ..................12
FIGURE 4: LAYERED ARCHITECTURE OF BIG DATA SYSTEMS (HU, ET AL., 2014) ...........................................13
FIGURE 5: WINDOWS DEFINE THE SCOPE OF OPERATIONS IN SPEs (STONEBRAKER, ET AL., 2005) ........17
FIGURE 6: THE THREE LAYERS OF THE LAMBDA ARCHITECTURE .................................................................18
FIGURE 7: BATCH AND REAL-TIME VIEWS UPDATES IN LAMBDA ARCHITECTURE (KINLEY, 2014) .........19
FIGURE 8: THE CORE PRINCIPLES OF REACTIVE APPLICATIONS (DEVORE, ET AL., 2015) ........................26
FIGURE 9: DIKW PYRAMID .......................................................................................................................................53
FIGURE 10: THREE DIFFERENT BIG DATA PROCESSING CONTEXTS .................................................................54
FIGURE 11: LAMBDA ARCHITECTURE IN EPIC REAL-TIME ..............................................................................55
FIGURE 12: QUERY REQUEST IN EPIC REAL-TIME ..............................................................................................56
FIGURE 13: EPIC REAL-TIME COMPONENTS ..........................................................................................................58
FIGURE 14: QUERY PROCESS FLOW (E.G. TRENDING HASHTAGS) .................................................................59
FIGURE 15: EVENTS SERVICE .....................................................................................................................................62
FIGURE 16: QUERY REQUEST .....................................................................................................................................64
FIGURE 17: QUERY ENGINE .......................................................................................................................................65
FIGURE 18: ACTORS AT RUN TIME ..........................................................................................................................67
FIGURE 19: REST API URL STRUCTURE ..................................................................................................................67
FIGURE 20: SPARK LAMBDA PROCESS ....................................................................................................................73
FIGURE 21: LAMBDA UPDATE ......................................................................................................................................78
FIGURE 22: EPIC REAL-TIME QUERY ENTRY FORM ...............................................................................................86
FIGURE 23: WEB UI FOR TREND HASHTAG QUERY ...............................................................................................87
FIGURE 24: WEB UI FOR POPULAR USER QUERY ...................................................................................................87
FIGURE 25: COMPONENTS OF IMMEDIACY TIME ..................................................................................................93
FIGURE 26: REAL-TIME QUERIES ON US PRESIDENTIAL ELECTION .................................................................96
Figure 27: Batch Queries on US Presidential Election ................................................................. 98

Figure 28: Epic Real-Time Horizontal Scaling ............................................................................... 100

Figure 29: A Proposed Extension to the Lambda Architecture ..................................................... 108
CHAPTER 1

INTRODUCTION

Real-time data analysis is fast emerging as an important service in the digital world. In the big data era—with data produced at high levels of volume, variety, and velocity—organizations want to exploit that data, to help them make strategic decisions, and to do so in real time. Recent interest in analyzing big data in real time has resulted in the active development of software frameworks and tools to perform real-time stream computation and analysis. However, there is still a need for robust software architectural designs that make these systems reliable, adaptable, and long-lived, while also increasing accessibility so they can be used by end users for customized analysis tasks.

Software engineers are more concerned than ever about issues of scalability, availability, and high performance. Data is now captured in vast volumes and speed from different sources. Social media platforms generate data in unprecedented amounts, and this creates new types of challenges for business and social analysts. I work in an application domain known as crisis informatics (Palen, et al., 2010); it is a field that has emerged to study the socio-technical, informational, and collaborative aspects of crisis events. Project EPIC at the University of Colorado Boulder—a leader in this research domain—has been studying Twitter data since Fall 2009 to uncover important social behaviors by members of the public during disasters.

Project EPIC’s analysts have been studying large social media data sets using batch processing techniques and now seek to make a transition to the real-time analysis of crisis data. Project EPIC’s software engineering efforts focused first on data collection, producing a system called EPIC Collect (Anderson & Schram, 2011; Schram & Anderson, 2012); those efforts then shifted to analysis, producing a system called EPIC Analyze (Anderson, et al., 2015; Barrenechea, et al., 2015; Aydin & Anderson, 2015). EPIC Analyze provides a variety of features to help Project EPIC analysts explore and analyze large collected Twitter data sets.

In this thesis, I present a system called EPIC Real-Time to help Project EPIC shift from batch processing techniques to analyzing data in real time. I performed an investigative study with
Project EPIC analysts (Chapter 3) to elicit their requirements with respect to performing crisis informatics research in real time. EPIC Real-Time is meant to enable real-time analytics on high volumes of Twitter streaming data by providing flexible queries, fast response time, reliable performance, and high availability.

Big data calls for new paradigms and system architectures for data processing. Big data analytics is an emerging area of research, defined as the process of applying exploratory analysis algorithms on big data to uncover hidden patterns or unknown correlations (Hu, et al., 2014). Big data analytics require different types of big data processing, categorized into two paradigms: real-time stream processing and batch processing. There are significant opportunities for real-time processing to reduce the costs of batch processing. However, when designing big data applications, one needs both batch and real-time to run in parallel. Therefore, engineers seek ways to combine real-time systems, like Apache Storm, with a Hadoop-based batch processing framework. This architectural combination of batch and real-time computation is known as the Lambda Architecture (Marz, et al., 2015), discussed in Chapter 2. Software engineers face many significant challenges when creating efficient, reliable, and scalable big data systems (Anderson, 2015).

My focus is on providing analysts with maximum flexibility with respect to the types of queries they can make on large data sets in real time as data streams in, as well as after the fact using batch processing techniques. Furthermore, I tackle many important issues related to developing highly scalable, reliable, and distributed software applications for big data analytics. I am embracing the philosophy of reactive programming (based on the Reactive Manifesto, http://www.reactivemanifesto.org) to influence how to design big data systems. The fundamental tenet of this approach is to build message-driven applications. This technique can lead to scalable, resilient, and responsive applications (DeVore, et al., 2015). I leverage the actor programming model provided by Akka to develop my message-driven system. With actors, I can build concurrent and distributed systems by sending and receiving messages asynchronously between actors whose locations are transparent to each other (Sathyanarayanan, et al., 2016).

The remainder of this thesis is organized as follows. Chapter 2 provides background information for this thesis. In Chapter 3, I provide the design for a prototype platform supporting big data analytics, along with my investigative user study. Then, I present the architecture and implementation of EPIC Real-Time in Chapter 4, and discuss the evaluation of my prototype in
Chapter 5. Chapter 6 reviews related work in the area of developing large-scale applications for real-time big data analytics. In Chapter 7, I identify some additional research directions for future work. Finally, in Chapter 8, I present my conclusions.
CHAPTER 2

BACKGROUND

In this section, I provide background information on the following topics: crisis informatics, Project EPIC’s existing software infrastructure, and various topics related to the field of big data.

2.1 Crisis Informatics Research

Over the last decade, the world has witnessed an unprecedented scale of crises experienced in a number of devastating natural disasters such as the major Haiti earthquake in 2010 and Hurricane Sandy in 2012. The world has also been suffering other types of man-made disasters such as political disruption, terrorist attacks, and health-related disasters, such as the spread of viral disease (Hagar, 2013). These kinds of dangerous events have motivated the formation of a new research field. Crisis informatics is an emerging interdisciplinary research area that studies “the interconnectedness of people, organizations, information, and technology during crises” (Hagar, 2013). Leaders in the field, such as Prof. Leysia Palen, describe crisis informatics research as taking an interdisciplinary perspective on the socio-technical, informational, and collaborative aspects of developing and using technologies and information systems in the disaster life cycle of preparation, warning, impact, response, and recovery (Palen, et al., 2007). Crisis informatics studies the social and behavioral effect of ICT (information and communications technology) during and after disasters, with the aim of analyzing and interpreting the roles ICT plays in crises. One of the main concerns of crisis informatics research is to study data generated from social media during mass emergency events (Hagar, 2013). The crisis informatics area has a rich literature of many research efforts dedicated to studying the qualitative and quantitative aspects of social data produced by members of the public during disasters.

2.1.1 Project EPIC: Current State

Project EPIC (Empowering the Public with Information in Crisis)—a large NSF-funded project (http://epic.cs.colorado.edu)—has been a leader in crisis informatics research. Project EPIC’s
researchers strive to analyze social media data to study different social behaviors during crisis, focusing on public needs, investigating how and why the public communicates, and looking for significant social patterns and valuable knowledge produced during different types of disasters. Through their work, they have defined a vision for studying the behavior of the public delivering, communicating, and mediating disaster-related information for large scale emergency response, considering not only official responders but also members of the public (Palen, et al., 2010). Project EPIC has enriched the crisis informatics field with many significant research contributions to help understand the socio-behavioral phenomena of self-organization of interested volunteers who mobilize online work or on-the-ground activities (Starbird & Palen, 2011; White, et al., 2014), information sharing between unofficial and official sources (Starbird, et al., 2010; Sarcevic, et al., 2012), medical coordination during a mass disruption event (Sarcevic, et al., 2012), or crowdwork and the great potential of crowd interaction in disaster response and recovery (White, et al., 2014; Palen, et al., 2015). While much has been investigated and published, research methods for collecting, storing, and analyzing social media data is still expensive and time-consuming. Massive amounts of social data need to be collected during and after crisis events, then preprocessed and persisted, and finally viewed, queried, and analyzed.

A research thesis by McTaggart, back in 2012, documented the research needs of analysts working at Project EPIC. Typically, when a disaster event unfolds, Project EPIC will start collecting tweets related to the event. Once analysts decide to stop collection for an event, data analysis begins. This includes statistical analysis and ad hoc methods of filtering and sampling. Research analysts report the need for a variety of analysis techniques required for a specific disaster event. Managing social media data collected during disasters was a significant challenge, and storing, viewing, filtering, and analyzing large data sets of tweets were carried out using traditional analytical tools like Excel and Project EPIC’s own eDataViewer (McTaggart, 2012). The limitations of those tools created a need for powerful data management systems to cope with the challenges of working with data sets consisting of millions of tweets.

**EPIC Collect** (Anderson & Schram, 2011; Schram & Anderson, 2012) is a system designed for reliable and scalable social media collection (Figure 1). When a new disaster event hits, Project EPIC analysts monitor Twitter for event-related keywords, and use the EPIC Event Editor—a simple web application—to associate those keywords with a new data collection. EPIC Collect connects to Twitter’s Streaming API to collect tweets containing those keywords and stores them
in a Cassandra cluster capable of storing terabytes of tweets. Analysts spend time monitoring and updating the event-related keyword lists, with the aim of collecting comprehensive Twitter datasets that include most of the relevant tweets posted during the corresponding crisis events. Software engineering efforts in Project EPIC were first invested to develop EPIC Collect to reliably provide 24/7 data collection. Then, the focus shifted to the challenges associated with data analysis leading to the design and development of EPIC Analyze.

**EPIC Analyze** (Anderson, et al., 2015) extended EPIC Collect to support social media analytics (Figure 1), providing a variety of features to help Project EPIC analysts explore and analyze previously collected Twitter datasets. EPIC Analyze is implemented as a Ruby on Rails web application used by analysts to browse, filter, and annotate large sets of tweets representing different disaster events. Datasets related to disaster events are migrated from the Cassandra cluster via batch-oriented scripts and then indexed by Solr before becoming ready for use within EPIC Analyze. Project EPIC’s infrastructure is aided by DataStax Enterprise (http://www.datastax.com/) which integrates Cassandra with many big data indexing and querying tools, such as Solr, Hadoop, and Pig. EPIC Analyze makes use of all these scalable tools, in addition to PostgreSQL—an object-relational database management system—to store comments and annotations made by analysts, and Redis—an in-memory key-value data store—to cache query results. EPIC Analyze is a reliable batch-oriented data analysis system. All query requests are processed quickly (in a few seconds, on average) since data is batch indexed by Solr when events are imported from EPIC Collect. One
reason for this speed is that once Solr has looked up the results of a query, it returns just the top 
50 results. More results are then available on additional pages of 50 tweets each or by refining the 
query and submitting again. EPIC Analyze’s querying process is designed to combine results from 
Solr and PostgreSQL (Barrenechea, et al., 2015); this allows tweets and the annotations made on 
them to be retrieved at the same time and displayed together. In addition to the entire range of 
query operations that Solr provides via its use of Lucene, EPIC Analyze also provides an advanced 
query interface that allows analysts to enable and disable parts of a query, jump back to previously 
submitted queries, and save the results of a query as a new dataset that can then be the target of 
further analysis (Barrenechea, et al., 2015). Another powerful feature of EPIC Analyze is its ability 
to incrementally sort large Twitter datasets. Sorting is done as a batch process that is triggered 
automatically after a set of tweets has been imported into EPIC Analyze. Tweets are sorted along 
a number of sort dimensions, with the results stored in Cassandra (for persistence across sessions) 
and in Redis (for access during a single session), that then allows EPIC Analyze to respond to sort 
requests in under a second independent of the size of a given data set. If the data set is still being 
actively collected, the sorting process is able to sort new tweets into the global sort order without 
having to re-sort previously sorted tweets (Aydin & Anderson, 2015).

2.1.2 Project EPIC: Next Generation

As discussed in the introduction, Project EPIC’s analysts require advanced data collection and 
analysis capabilities to allow them to study mass emergency and mass convergence events. 
However, to ask questions in real-time, analysts need to perform the data analysis process of sense- 
analyze-act in seconds and minutes instead of days and weeks (Figure 2). Project EPIC is making 
this transition now, partly through the research presented here. My goal is to augment the analysts’ 
capabilities such that they can quickly navigate the digital space surrounding a crisis event and 
allow them to hunt for related information. Furthermore, the ability to specify queries is paramount 
and so much of my work will focus on providing flexible query support in both real-time and 
batch-processing contexts.
Based on these needs, new technologies and tools need to be incorporated to support the next generation of researchers at Project EPIC and to identify lessons learned that can be used to apply these techniques to many other problem domains that require real-time data analysis. Having presented information concerning crisis informatics and Project EPIC, I now turn my attention to providing background information about various topics related to the field of Big Data Analytics.

2.2 Big Data Analytics

2.2.1 Big Data: An Emerging Paradigm

The term “big data” was coined to capture the work being performed in many disciplines to address the trend of sensors, devices, and software being able to generate and analyze large amounts of data. Big data calls for new paradigms and system architectures for data processing (Hu, et al., 2014). Big data refers to large or complex data that traditional data management and analysis systems are inadequate to handle. The term “big data” defines also the emergent paradigm of collecting, processing, analyzing, and storing extremely large data sets usually at a scale of petabytes and beyond. Formally, big data is characterized by the “5Vs” model (Chandarana & Vijayalakshmi, 2014): Volume, Velocity, Variety, Veracity, and Value. These different dimensions of big data create significant difficulties and complexities which traditional relational database management systems (RDBMs) struggle to address.
The last decade has seen the advent of software and hardware technologies that can handle massive volumes of continuously growing structured and unstructured data; these technologies help solve some of the challenges associated with designing and developing data-intensive software systems. Software engineers face many significant challenges when creating efficient, reliable, and scalable data-intensive software systems (Anderson, 2015). These challenges are normally related to capturing, storing, analyzing, and visualizing large data sets. One useful class of technology to help address some of these challenges are *NoSQL data stores* (Schram & Anderson, 2012; Sadalage & Fowler, 2012). These systems can scale to vastly larger sets of data, without imposing a schema, solving many problems of traditional RDBMs; examples of NoSQL systems include Cassandra, Solr, Redis, HBase, Riak, and MongoDB (among many others). The past decade has also seen a huge amount of innovation in large-scale distributed computation systems, such as Hadoop. Apache Hadoop provides an approach to computation known as MapReduce (Lam, 2010) that works well for processing large data sets on clusters of computers. MapReduce is a batch processing mechanism that is best suited for long-running, background processes. Hadoop also provides reliable distributed storage of data via HDFS. The advantage of Hadoop is that it provides a low-cost solution to the reliable batch processing of large data. Analytical tools have then been added to this ecosystem like Hive and Pig.

Hadoop can handle data volume and variety but struggles with data velocity. Traditionally, data is stored to be analyzed in an off-line manner; however, many applications such as fraud detection, trending topics calculation, or online social interactions require on-line processing to give real-time or near real-time responses to queries. In these situations, stream processing is used. These frameworks can compensate for the high-latency of batch processing systems. The demand for stream processing is increasing since just processing large data sets is not enough. Data has to be processed quickly, so that an organization can react to changing business conditions in real time. As a result, a high percentage of organizations are moving or planning to move to real-time big data analytics. Although Hadoop can be configured to take advantage of processing data in memory to get faster response times, it was never designed to handle streaming data. As such, there is an emerging wave of streaming analytics tools like Apache Storm and Apache Spark to address this need. Integration of these different big data systems is not trivial, but, if done correctly, organizations can realize the benefits of both approaches to analysis. In big data contexts, different types of processing are required to run different types of analytics. Often, a combination of batch
processing, online transaction processing, interactive ad hoc query processing, and stream processing is observed (Chandarana & Vijayalakshmi, 2014). For the rest of this section, I provide an overview on different big data paradigms, big data system architectures, big data analytics, stream processing, the Lambda Architecture, and Apache Spark.

### 2.2.2 Big Data Paradigms: Batch Vs. Streaming

Big data analytics is an emerging area of research, defined as the process of applying exploratory and analysis algorithms on big data to uncover hidden patterns or unknown correlations (Hu, et al., 2014). Big data analytics require different types of big data processing, categorized into two paradigms, stream processing and batch processing.

**Batch Processing:** In the batch-processing paradigm, data are first stored and then the entire dataset is scanned to produce an analysis result. Much time is wasted during data transmission, storage, and repeated scanning. MapReduce has become the dominant batch-processing model.

**Stream Processing:** This paradigm analyzes data as soon as it arrives, applying what is called “continuous queries” to the data stream. Because this stream is fast and carries enormous volume, only a small portion of the stream is stored in memory while it is processed. Representative open source systems for stream processing include Storm, S4, Spark Streaming, and Kafka.

There are significant opportunities for the real-time processing paradigm to reduce the overhead costs of batch processing. Incremental computation in stream processing attempts to analyze only the most recently added data and combines that analysis with the previously calculated state to produce a result. However, when designing big data applications, one needs both batch and real-time to run in parallel. Therefore, engineers seek ways to combine real-time systems, like Apache Storm with Hadoop-based batch processing frameworks. This architectural combination of batch and real-time computation has come to be known as the Lambda Architecture (Marz & Warren, 2015), discussed in Section 2.2.6.

The abstraction for a data flow is called a *stream*, which is an unbounded sequence of tuples. A *tuple* is a structure for wrapping up a set of data values into a single *event* that needs to be processed. There are two different paradigms for big data real-time stream processing: tuple at a time and micro-batching (Shahrivari, 2014).
**Tuple at a Time:** This kind of processing is true to the definition of stream processing as it processes every tuple as it arrives and attempts to do so in milliseconds. Apache Storm implements this approach.

**Micro-Batching:** The idea of micro-batching is to apply the well understood concept of batch processing to streaming data by keeping the batch sizes very small. A microbatch is a stream of all the events that have arrived within the last few milliseconds. A microbatch could also be a fixed number of collected events. Apache Spark’s Streaming library implements this approach.

### 2.2.3 Big Data System Architecture

In this section, I discuss big data system architectures from different points of view.

**Big Data System: A Value-Chain View**

A big-data system is a complex software system, handling the lifecycle of big data from its birth until its final representation. A typical big data system involves multiple phases for different types of functionality. Adopting a systems-engineering approach, a typical big data system is decomposed into four consecutive phases, including data generation, data acquisition, data storage, and data analytics (Hu, et al., 2014). This big data timeline is shown in Figure 3, showing every phase with the corresponding big data technologies used to address that phase. *Data generation* concerns how data is generated and what kinds of data sources are needed, such as sensors, log files, web crawlers, business transactions, social media, health and scientific data, etc. *Data acquisition* refers to the process of obtaining information; it is subdivided into data collection, data transmission, and data pre-processing. *Data collection* is dedicated to acquiring raw data from its different production sources. *Data transmission* requires high-speed data transfer mechanisms to move data to subsequent processing. *Data pre-processing* is responsible for cleaning data of any noise or duplication as dictated by data quality requirements. *Data storage* concerns persistently and reliably storing large-scale datasets by adopting SQL or NoSQL databases technologies or distributed file systems while using sophisticated data management systems to enable interactive querying of the stored data. The last stage is *data analysis* which leverages analytical methods to analyze and model data to extract value.
Big Data System: A Layered View

A big data system alternatively can be decomposed into three layers, illustrated in Figure 4, including the infrastructure layer, computing layer, and application layer (Hu, et al., 2014). The **infrastructure layer** consists of different technologies of physical storage, computation, communication, and cloud computing. The **computing layer** includes data integration, data management, and programming models. **Data integration** means acquiring different data sources and integrating them into unified forms. **Data management** refers to different data management tools to efficiently interact with data, such as distributed file systems and SQL or NoSQL databases. The programming model represents any application logic that facilitates data analysis, such as the MapReduce model.

Figure 3: Big Data Technology Map for the Big Data Value Chain (Hu, et al., 2014)
2.2.4 Data Analytics: Common Methods and Types

Big data analytics creates another class of challenges exploring and analyzing big data that requires techniques from many fields such as statistics, machine learning, network science, and data mining. The benefit associated with big data analytics stems from its capabilities to produce useful values, uncover hidden patterns, suggest conclusions or recommendations, or support decision-making. Big data analytics relies on obtaining raw data about an event of interest through observation, measurement, or experiments (Hu, et al., 2014). With the flood of data available to businesses, companies are turning to analytics solutions to extract meaning from the huge volumes of data to help improve decision making. Big data analytics can be applied to disease prevention, traffic routing, security breach detection, product enhancement, supply chain optimization, and many other important applications. IBM provides insights on “giving data meaning—despite its growing volume, variety and velocity”, and how important it is for organizations to incorporate analytics solutions with different capabilities for analyzing historical and real-time data (IBM white paper, 2013). A current focus for big data analytics is developing technologies that can process even higher volumes of streaming, heterogeneous data.
There are common methods for big data analytics, regardless of application domains: data visualization, statistical analysis, and data mining (Hu, et al., 2014).

**Data visualization** involves using different types of graphics to present analysis results. In general, bar charts and maps help people see and understand information easily. Data visualization is important to big data analysis as it helps quickly deliver facts and uncover hidden patterns. Big data requires advanced visualization tools and libraries that cope with large amounts of data, such as Tableau Software (www.tableau.com), and D3 (d3js.org).

**Statistical analysis** is based on statistical theories that can be applied to big data sets. Descriptive statistical analysis can summarize large sets of data, whereas inferential statistical analysis can draw inferences about data, including testing hypotheses and deriving estimates.

**Data mining** is the computational process of discovering patterns in large data sets. Classification, clustering, regression, and statistical learning techniques are the most important topics in data mining research. Different data mining algorithms have been developed in the artificial intelligence, machine learning, pattern recognition, statistics, and database communities, including K-means, Neural Networks, Decision Trees, SVMs (support vector machine), EMs (expectation maximization), PageRank, Naive Bayes, and CART.

Big data analytics is also classified with respect to the depth of analysis performed as characterized by descriptive analytics, predictive analytics, and prescriptive analytics (Hu, et al., 2014; Bertolucci, 2014).

**Descriptive Analytics** is the simplest type of big data analysis. The purpose of descriptive analytics is to describe or summarize raw data and make it something that is interpretable by humans. Descriptive analytics uses data aggregation and data mining techniques to provide insight into the past and answer: “What has happened?” The past refers to any point of time that an event has occurred, whether it is one minute ago or one year ago. Descriptive analytics is useful because it allows one to learn from past behaviors and then prepare for the future. More than 80% of business analytics and social analytics are descriptive (Bertolucci, 2014). Descriptive analytics in business describe a company’s production, financials, operations, sales, inventory and customers. For web and social analytics, there are thousands of metrics; for example, the number of posts, mentions, followers, page views, pins, etc. in a given data set.
**Predictive Analytics** is the next step. Predictive Analytics uses statistical modeling techniques to predict future probabilities and trends to answer: “What could happen?” The basis of predictive analytics is working with data you have to predict data that you do not have. This works by applying statistical techniques such as linear and logistic regression to understand trends and predict future outcomes and using data mining to extract patterns to provide insight and forecasts. Sentiment analysis is a common type of predictive analytics, where one takes plain text as an input to a model, producing a sentiment score (positive/negative) as an output.

**Prescriptive Analytics** is the third and final phase of big data analytics; it helps to support decision making. It uses optimization and simulation algorithms to provide advice on possible outcomes and answer the question “What should we do?” Prescriptive analytics includes descriptive and predictive analytics, but goes beyond by recommending one or more courses of action, and showing the consequences of each decision. Prescriptive analytics not only answers “What will happen?” and “When will it happen?” but also “Why will it happen?” This approach to analytics allows business decision-makers to act based on an optimal set of actions and constraints. Prescriptive analytics uses a combination of techniques and tools such as business rules, algorithms, machine learning, and computational modeling procedures. Prescriptive analytics is powerful but relatively complex to administer in business or scientific domains.

All of these analytical techniques can be applied to many different types of big data including historical data and real-time streaming feeds. Yet, it is important for an organization to match its infrastructure, technologies, and processes to the stage of analytics it needs to perform and the goals it wishes to reach, while at the same time, investing in more advanced technologies and processes to support more complex analytics (IBM white paper, 2013).

### 2.2.5 Stream Processing

Stream processing is a technology that allows for the collection, integration, analysis, visualization, and system integration of data, all in real time, as data is being produced. Stream processing is typically performed on streaming data while it is held in memory (i.e. not on disk); the events in the stream are typically unstructured but including at the very least a timestamp that indicate the time the event was created or arrived. Stream processing has great potential to extract actionable intelligence, and to react to operational exceptions creating real-time alerts and
automated actions. Twitter’s Storm, Yahoo’s S4, Cloudera’s Impala, and Apache Spark, are all examples of stream processing technology (Liu, et al., 2014).

Stream processing is a very active area of research, commercial, and open-source development. Research on stream processing emerged in the late 1990s with the development of several early prototypes, including academic (e.g. STREAM, StreamIt, Aurora) and commercial platforms (e.g. InfoSphere, StreamBase). Stream processing systems originated from the early development of information flow processing—defined as “an application domain in which users need to collect information produced by multiple, distributed sources, to process it in a timely way, in order to extract new knowledge as soon as the relevant information is collected (Cugola & Margara, 2012).” Information flow processing can be grouped into four broad classes: active databases, continuous query systems, publish-subscribe systems, and complex event processing systems. All of them are predecessors that have helped shape stream processing (Andrade, et al., 2014).

Stream processing engines (SPEs) are designed to cope with real time data stream processing. SPEs use a graph oriented paradigm, representing operators as vertices and edges as flows of data. This graph structure allows SPEs to cope with thousands of events per second. Solutions such as Apache Storm, D-Stream, Samza, and Kinesis operate under this common basis where operators such as filters, aggregators, and counters, are organized on a graph and they are distributed over the nodes to be processed. SPEs perform SQL-style processing on incoming data without necessarily storing them. Yet, to store state when necessary, an embedded SQL database can be used for efficiency. SPEs use specialized primitives and constructs (e.g., time-windows) to express stream-oriented processing logic and define the scope of a multi-message operator. Such windows are designed to slide a predefined amount from the current window. Based on chosen window size and slide parameters, the sliding of windows can cause disjoint or overlapping between windows (see Figure 5; Stonebraker, et al., 2005).
Figure 5: Windows Define the Scope of Operations in SPEs (Stonebraker, et al., 2005)

Existing SPEs for real-time or stream data processing like Apache Spark Streaming or Apache Storm create a basis for research and hypothesis testing of data streams. These technologies are fault-tolerant, scalable, and allow for code in several programming languages like Python, Java and Scala. Ideally, stream processing applications need to satisfy several performance requirements, such as low latency and high throughput, long-term high availability, and faster information analysis and discovery (Liu, et al., 2014; Andrade, et al., 2014).

Stream processing and its ability to process data online supports the needs of continuous big data analysis. Furthermore, massive amounts of historical data must still be maintained, either because one wants to analyze and query older data in conjunction with recent data, or because one wants to inject new analytical techniques that were not available when some of the older data was collected (Andrade, et al., 2014).

Continuous Queries: Stream processing is designed to analyze and act on real-time streaming data using *continuous queries*. A continuous query is a persistent, standing query that is not performed once, but permanently installed and continuously executed over streaming data, until it is explicitly uninstalled. As a result, a continuous query produces new results as long as new data arrives at the system. Continuous queries are similar to database queries in how they analyze data, but they differ by operating continuously and updating results in real-time. Sources can include any type of data collected in or generated within an enterprise (Andrade, et al., 2014). The concept of continuous queries creates the foundation for complex event processing.

Complex Event Processing: Complex event processing (CEP) enables the aggregation and analysis of events from a variety of sources to support the detection of complex events via a large set of business rules. In CEP systems, events generated by multiple sources are collected, filtered,
aggregated, combined, correlated, and dispatched to analysis applications in real-time. In this way, applications can continuously react to information embedded in the incoming data. These systems typically associate a set of semantics with the data stream, based on a set of rules or patterns expressing the event detection task. Complex events are detected by first detecting sequences of simpler events (Cugola & Margara, 2012). CEP systems also have limitations. First, most CEP systems are centralized that cannot scale when working with large data sets. Second, the capability of programming models in CEP systems are usually limited for big data applications, in which one needs dynamically modified models to process different types of data using different exploratory tasks (Andrade, et al., 2014). The research of leveraging CEP in a big data context is still immature, yet its addition to a big data system can lead to more sophisticated real-time big data analytics.

### 2.2.6 The Lambda Architecture

The Lambda Architecture was recently proposed by Nathan Marz as a software architecture for big data systems that perform both real-time analysis and batch processing (Marz & Warren, 2015). The Lambda Architecture is split into three layers: the batch layer, the serving layer, and the speed layer (Figure 6).

![The Three Layers of the Lambda Architecture](image)

**Figure 6: The Three Layers of the Lambda Architecture**

**Batch layer:** The batch layer is responsible for storing the master copy of the dataset and precomputing batch views on that dataset, usually as Hadoop jobs. Computing the views is a
continuous operation, so when new data arrives, it will be aggregated into the views when they are recomputed during the next iteration.

**Speed layer:** The speed layer is responsible for computing real-time views from the data it receives using an incremental model. The speed layer is needed to compensate for the high latency of the batch layer by computing real-time views. The real-time views contain information on the data streaming in and not on the entire data set. These views help to supplement the views generated by the batch layer.

**Serving layer:** The views from the batch and speed layers are sent to the serving layer. The serving layer is responsible for merging, indexing and exposing the merged views so that they can be queried. Since the batch views are static, the serving layer only needs to provide batch updates and random reads. As the real-time views are incremental, the speed layer requires both random reads and writes to continuously increment the real-time views.

Interestingly, real-time views are intended to be transient, as they can be discarded, as soon as a copy of these views are propagated into the batch layer. This is referred to as “complexity isolation” (Marz & Warren, 2015), meaning that the most complex part of the architecture is pushed into the layer whose results are only temporary, as demonstrated in Figure 7.

![Figure 7: Batch and Real-Time Views Updates in Lambda Architecture (Kinley, 2014)](image)

This architecture defines the most general system of running arbitrary functions on arbitrary data using the following equation “query = function (all data)”. The premise behind the Lambda Architecture is that it runs ad hoc queries against all data, with the trade-off that queries will be out of date by a few hours. The serving layer updates whenever the batch layer finishes precomputing a batch view. This means that the only data not represented in the batch
view is the data that came in while the batch computation was running. This missing update in batch views is compensated by the real-time views.

The Lambda Architecture enables building large-scale, distributed data processing systems in a flexible and extensible manner, providing fault-tolerance both against hardware failures and human errors. By replacing existing precomputed views and re-computing the entire output, all errors will be fixed periodically. Furthermore, future algorithms or analytics can be applied to all the data by simply adding another view on the whole data.

However, developing and maintaining an application that makes use of the Lambda Architecture is expensive. Implementing the batch and real-time views in one system is complex; it typically requires two different underlying systems. For example, the batch views might be implemented using Hive, while the real-time views are implemented in Storm. As a result, business logic is duplicated in two places, producing two different codebases, each needing to be implemented and maintained separately.

Luckily, Apache Spark offers a simple and elegant solution. Spark provides a technology stack that allows developers to implement a Lambda Architecture compliant system using a unified development and test environment, while supporting big data analytics in both batch and streaming configurations (Shapira, 2014).

2.2.7 Apache Spark

Apache Spark (https://spark.apache.org/), an open-source distributed computation framework, has recently gained significant momentum in the big data community and is a promising alternative to Hadoop-based MapReduce. Spark, originally developed at UC Berkeley in 2009, is a fast, general engine for large-scale data processing, bringing high-speed, in-memory analytics to Hadoop clusters, or by running on a stand-alone cluster. It offers interactive code execution using Scala, Python, and Java, alongside a rich set of libraries.

Spark comes packaged with a powerful set of libraries, including support for SQL queries (Spark SQL+DataFrames), streaming data (Spark Streaming), machine learning (MLiB) and graph processing (GraphX). Thus, it can support complex analysis workflows. From an architecture perspective, Spark is based on two key concepts: resilient distributed datasets and a directed acyclic graph execution engine.
A directed acyclic graph (DAG) engine in Spark supports cyclic data flow and in-memory computing. Each Spark job creates a DAG of task stages to be performed on the cluster. Instead of defining two stages of “Map and Reduce,” DAGs created by Spark can contain any number of stages. This allows some jobs to complete faster than they would in Hadoop. Simple jobs can be completed after one stage, while complex tasks can be completed in a single run of many stages, rather than having to be split into multiple jobs.

Resilient distributed datasets (RDD) is a fundamental data structure of Spark, representing a collection of data partitioned across machines (Zaharia, et al., 2010). Any function applied on an RDD is called a transformation. Transformations are functions that create a new RDD from an existing one. All transformations in Spark are lazy. They do not compute their results once defined. Instead, they remember the transformations applied to some data then compute when an action requires a result that needs to be returned to the application (called the driver). Spark Streaming provides a high-level abstraction called a discretized stream or DStream, that represents a continuous sequence of RDDs. DStreams are created when Spark receives input data streams or applies transformations on existing DStreams (Zaharia, et al., 2013).

2.3 Modern Software Development Trends

In the Big Data era, enterprises have to rethink their traditional approaches to software development. They need to break up their information silos and integrate with other data systems to create efficiency in business processes. They need to maintain data systems that avoid the complexity of traditional techniques. They need to design for scalability with big data. Marz discusses the desired properties of a big data system: robustness and fault tolerance, low latency reads and updates, scalability, generalization, extensibility, ad hoc queries, minimal maintenance, and support for debugging (Marz & Warren, 2015). To provide these desired properties, one needs to adopt long-term flexible development strategies.

An approach that makes use of agile life cycles is essential for big data development; it affords architects and developers the freedom to rapidly support a wide variety of changing business requirements and needs. When flexibility or agility in business is a high-priority, service-oriented computing is an approach to the design of software systems that offers significant benefits. For the last decade, the adoption of a service-oriented architecture (SOA) as an underlying approach to
Software design has been gaining popularity and widespread use. Internet-based enterprises, such as Amazon, have embraced SOA paradigms, since monolithic applications were no longer fit to the exponential growth they endured of distributed applications and users. Service-oriented architectures, software as a service, cloud computing, and many other modern solutions have all played roles in helping enterprises achieve greater application integration. In fact, the interoperability of modular components provided by SOA is important for the success of big data architectures. Modular applications based on interoperable, loosely coupled components are easier to maintain and upgrade than large monolithic applications (Krause, 2015). A service-based approach, or the recent microservices approach, can help to achieve this type of loosely coupled, highly interoperable system architecture.

SOA remains a solid foundation for enterprise applications, and has diverged to include the design of APIs and microservices. APIs help with the consumption and distribution of business capabilities as sharable services, and microservices deliver flexibility and scalability to the development and deployment of service-based applications. An emerging software development paradigm is to build web-scale, distributed, and decoupled applications based on microservices. New features in development to tame software complexity include microservices, API management, platform-as-a-service, in-memory databases, and containerization (Paul, 2015).

Furthermore, the requirements of modern applications have changed dramatically in recent years. Today, applications are deployed on hundreds or thousands of servers leveraging cloud-based clusters, serving data in petabytes, that is delivered concurrently to millions of users on mobile devices, with millisecond response time and 100% uptime. These systems need to be more robust and flexible to meet modern software demands. More specifically, large-scale systems need to be responsive, resilient, elastic and message driven. These systems are called Reactive Systems (DeVore, et al., 2015; Sathyanarayanan, et al., 2016). Reactive programming is an attempt to implement all these features in a new generation of software. This is influenced by the Reactive Manifesto (http://www.reactivemanifesto.org) that outlines a set of desired properties to build into applications that encounter data growth in the era of big data. I discuss reactive programming with some details further in the coming sections with some emphasis on the promise of the actor-model for building reactive applications. I discuss many of these concepts in more detail next.
2.3.1 Microservices

Microservices, as defined by Fowler and Lewis is “an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API” (Fowler & Lewis, 2104). Accordingly, the idea behind microservices is to decompose large and complex applications into a set of cohesive services that evolve over time. The word “micro” in microservices strongly suggests that the services should be small to promote the fact that each service should do one thing very well (Richardson, 2014; Krause, 2015). Adopting microservices can lead to reduced code/logic duplication, increased cohesion and lower coupling between the parts of a system. This shift, in turn, should create more scalable and easier to change software applications.

Microservices and SOA

The ideas of SOA have been welcomed and promoted in a large number of enterprises. However, there has not been widespread success for SOA because the overhead was too high to adhere to its many principles, governance, and protocols, such as SOAP and XML. The author of a microservices book (Krause, 2015) argues that microservices inherit the design principle of SOA but remove the baggage of WS* and heavyweight ESB. Fowler also had similar arguments because microservices are similar to many practices of SOA (Fowler & Lewis, 2104). However, some microservice advocates reject a connection to SOA entirely, while others consider microservices to be a lightweight or fine-grained SOA. I believe the concept of service orientation unifies microservices and SOA. Wikipedia defines service orientation as “a design paradigm for computer software in the form of services.” These services should be fine-grained, decoupled, isolated, deployable, and scalable. Despite not being an entirely novel concept, microservices are still worthy of exploration since they have the potential to solve many of the application design and architectural problems that plague software organizations.

Monolithic Applications

To adopt the microservices paradigm, organizations need to move away from monolithic applications, i.e., single-tiered applications running on a single platform. Even with a logically modular design, a monolithic application is deployed as a single component. An example with respect to Java applications is a single WAR file running on a web container such as Tomcat. On the positive side, monolithic applications are simple to develop since most IDEs are oriented
around developing single applications and deployment itself is also simple. One just needs to send a copy of the archive to an appropriate server. The monolithic architecture works well for small applications, but becomes complicated and unmanageable for complex applications. It is difficult to apply frequent changes and deployments, difficult to adopt new technologies, and difficult to scale to build large long-lived applications (Richardson, 2014).

**Benefits of Microservices**

Microservices provide many benefits. First, small services are easier to understand, making developers more productive and deployment straightforward. Second, services are deployed and scaled independently. Third, system can be more resilient. If one microservice fails, only a small bit functionality is lost. It is more simple to build system resilience around a smaller service. Finally, the ability to use different languages and frameworks eliminates any long-term commitment to an old technology stack (Richardson, 2014; Krause, 2015).

**Drawbacks of Microservices**

Since no technology is a silver bullet, microservices have a number of significant drawbacks that do not apply to monolithic applications. First, there is additional complexity when creating a distributed system that involves using some mechanisms for inter-process communication. Second, writing automated tests that cover multiple distributed services is difficult. Third, when deploying features that span multiple services, managing dependencies between services should be done carefully. Finally, managing multiple instances of different types of service requires a high-level of automation (Richardson, 2014; Krause, 2015).

**Inter-Service Communication Mechanisms**

The communications between microservices is different than in monolithic applications where regular method calls are used. Microservices must use inter-process communication. There are two main approaches to inter-process communication in a microservice architecture. The first option is to use synchronous HTTP-based mechanisms such as REST or SOAP. This is a simple request/reply style of communication over the internet, however, both the client and the server must be simultaneously available, which is not ideal in a distributed systems context. The other option is to use asynchronous message-based mechanisms, such as an AMQP-based message broker, in which message brokers buffer messages until consumers are ready to consume them.
One of the advantages here is that the message producer is decoupled totally from the message consumer, however, message brokers can add more complexity to a system (Richardson, 2014).

### 2.3.2 API Management

API management is the process of publishing, promoting, and overseeing application programming interfaces in a secure, scalable environment. The goal of API management is to allow an organization that publishes an API to monitor the interface’s life cycle and make sure the purpose of using the API is being met. API management software can be built in-house or purchased as a service through a third-party provider, such as 3Scale (3Scale, 2011), Apigee (Anuff, 2015), and API Axle (http://apiaxle.com). API management software in most cases provides complete solutions for creating and managing APIs, publishing APIs, scalable routing and monitoring traffic, and securing API content. As mentioned, the growing API movement is part of an ongoing shift in SOA moving more toward agility and simplicity and away from complexity and formalization, favoring more lightweight JSON and REST services.

### 2.3.3 Reactive Programming

There has been a significant shift in recent years toward reactive programming. This was documented when organizations tried to find common solutions and patterns for developing quality software in a modern software development context that needs to handle massive growth in both data and users (Reactive Manifesto, 2014, http://www.reactivemanifesto.org). The fundamental principle of reactive programming is to build message-driven applications. This technique can lead to scalable and resilient applications (DeVore, et al., 2015). The ultimate goal is to build responsive applications. All of these properties depend on the use of a message-driven architecture. The properties of reactive applications are required to meet the expectations of today’s users. Figure 8 shows the tenets of reactive applications and their relationships.
Responsiveness is a crucial factor to ensure a positive user experience with our systems. This property refers to a system’s ability to respond in an acceptable time to a user’s request. A responsive application needs to be able to provide fast service even when things go wrong, such as when some aspect of the system has failed or the entire system is experiencing a spike in network traffic (Sathyanarayanan, et al., 2016).

A resilient application is one that can continue to operate despite the presence of failures. Furthermore, it can be repaired quickly either via automated recovery procedures or via repairs instigated by human operators. The use of horizontally-scalable persistence frameworks can increase a systems resilience since these systems can automatically replicate information to guard against hardware failure or network congestion.

The use of a message-driven software architecture also supports building resilient applications, which, in turn, support the creation of responsive applications. The use of messages provides isolation which allows a system to recover after a failure. Any messages received for a component that failed are queued waiting for that component to return. This guarantees that the failure of one component in a system will not unduly impact the overall responsiveness of a system. On the other hand, location transparency in message-driven architectures allows different components on different clusters to interact transparently with each other (Webber, 2014). In my work, I demonstrate the use of a message-driven architecture via my use of Akka’s supervisor hierarchies and Akka Clustering that allow the locations of my actors to be transparent from one another.
A responsive application also needs to be scalable. A scalable system can dynamically respond to situations of higher load, automatically adding resources, such as servers and storage, to handle a usage spike. When designing a system to be scalable, there are two different methods that can be used. *Scaling up* (a.k.a. vertical scalability) involves making use of more powerful hardware. *Scaling out* (a.k.a. horizontal scalability) involves adding more cheap commodity nodes to a system, such as adding a new node in a cluster for a distributed software application. This is cost efficient, however, not all systems support it. For achieving scalability, ideally, the paradigm is first selected, and then the languages and software frameworks that embrace that paradigm are identified. These difficult technical decisions also involve selecting a concurrency programming model. There are two distinct concurrency models that can be embraced: thread-based concurrency and message-driven concurrency (Webber, 2014).

Message-driven architectures provide loosely coupled, asynchronous, and non-blocking applications, in which asynchronous boundaries decouple system components from *time and space*. This provides abilities to easily scale out on demand. This is also known as *elasticity*—defined as the ability to add new resources to a system when needed. As a result, we cannot build reactive applications using thread-based frameworks, because these frameworks are difficult to scale out, since they are built based on shared mutable state, threads, and locks. Amazon is a perfect example of on-demand use of hardware and services. When load is low, it spins down services, and when traffic spikes, it spins services back up (DeVore, et al., 2015). A message-driven application may be event-driven, actor-based, or a combination of the two. An *event-driven* system is based on events with some subscribed observers. An *Actor-based* system makes use of a message-passing architecture, where messages are sent between actors. Such messages may cross machine boundaries to reach actors on different physical servers. This feature enables scaling out on demand by allowing actors to be deployed across a cluster (Webber, 2014).

### 2.3.4 The Actor Model

The Actor model provides a high level of abstraction for writing concurrent and distributed systems. Actor-based applications are built based on asynchronous message passing between multiple actors. An *actor* is a light-weight programming construct embodying processing, storage, and communication. Every actor has a *mailbox* for receiving messages. An actor’s *logic* is used to determine how to handle each type of message it receives. Each actor has *isolated unshared*
state for storing context between requests (Roestenburg, et al., 2016; Webber, 2014; Sathyanarayanan, et al., 2016). Actors can pass messages to other actors, or even pass messages to themselves. Actor-based concurrency provides high levels of scaling computation out across network boundaries. This is a powerful concept that builds scalable applications that are also easy to design, build, and maintain. Another major benefit of an actor-based architecture is the loose coupling of actors. Actors do not block a thread waiting for a response, therefore actors can work on other tasks while a response is received asynchronously. Akka (akka.io) is an actor-based toolkit and runtime for building highly concurrent, distributed, and fault tolerant actor-based applications on the JVM. Akka has a number of other features for building reactive applications, such as supervisor hierarchies (Webber, 2014).
CHAPTER 3

ANALYSIS AND DESIGN

In this thesis, I investigate the design space of software systems for big data analytics by designing and implementing a new platform—EPIC Real-Time—via scalable distributed REST-based services to support analysts working in the crisis informatics field. The new platform is designed and built based on the Lambda Architecture. While the Lambda Architecture specifies various desirable aspects of a data intensive software system, it leaves many design tradeoffs and implementation details unspecified. This under-specification leaves room for innovation. My work in this thesis is driven from the need to support the analysts of Project EPIC in their transition from studying crisis events months after they occur to developing research questions and gathering data to support them while an event unfolds. Through this thesis, I present a prototype system that has both batch processing and stream processing capabilities and provides a set of services and applications that meet the diverse needs of Project EPIC analysts. While my prototype addresses the design challenges associated with providing comprehensive data analysis for crisis informatics researchers, my results are also applicable for other problem domains.

3.1 Research Questions

My specific research questions are:

1. What query capabilities and support tools need to be provided to analysts to allow them to perform crisis informatics research in real-time?

2. What software architecture and development methods are needed to implement these features such that it is possible to execute fast, reliable, and scalable analytic queries in real-time?

3. How can these real-time goals be achieved while also providing analysts with access to all of their historical data sets? How might the Lambda Architecture support this goal and what costs and challenges are associated with designing and implementing a software system that follows the Lambda Architectures rules and constraints?
3.2 Investigative User Study

I conducted an investigative study to serve as input to my research. The study investigated the requirements and needs of crisis informatics researchers when analyzing large crisis datasets in real time. I interviewed seven analysts associated with Project EPIC, each active in conducting research on crisis informatics. These analysts have a current set of research practices for analyzing crisis data after its associated event is over. The goal of my interviews was to get these analysts to reflect on their current research practices and how those practices might change if they were able to perform their analysis in real-time while a crisis event unfolds. Since each analyst works on different research problems, I interviewed them separately to collect their specialized set of requirements. As a result, this investigative study provides an overview of analysts needs across a broad spectrum of crisis informatics research topics; this information can be used to determine a general set of analytical methods that can be provided by my proposed platform. The interviews consisted of asking each analyst seven questions designed to elicit information about their current practices and their thoughts on real-time analysis. Those questions are shown in Table 1. Below, I report on each of the seven interviews.

<table>
<thead>
<tr>
<th>Interview Questions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. How do you currently conduct crisis informatics research?</td>
</tr>
<tr>
<td>2. What kind of data analysis and queries do you currently perform in this research?</td>
</tr>
<tr>
<td>3. How might your current techniques need to change when working in real-time?</td>
</tr>
<tr>
<td>4. What types of views and output would you like to use when performing real-time analysis?</td>
</tr>
<tr>
<td>5. What kind of research decisions do you think you might make as an event unfolds?</td>
</tr>
</tbody>
</table>
What would be the benefits of incorporating real-time analysis into your research?

What challenges do you foresee dealing with real-time analysis in your research?

Table 1: Interview Questions

3.2.1 Individual Interviews

Analyst 1

First Question: Current Research Practice

Analyst 1 is an active crisis informatics researcher focusing on how people mobilize in digital spaces in disaster response and what their intrinsic and extrinsic motivations are around what activities they take on in response to a disaster. She is interested in understanding human actions in disaster by examining the effects of environment, varying skills, and the problems experienced in a socio-technical cooperative work environment (in this case, a disaster event). Specifically, she is interested in investigating the self-organizing of online pet advocates. She did many ethnographic studies to examine how crowd work emerges to form distributed cooperative work, by investigating, for instance, dedicated Facebook pages, collecting Timeline Posts, Album Photos, Likes, and Comments posted in that page for a seven-month period. She also emphasizes the relationships between online work and offline or on-the-ground activities and how they afford and support mobilization of interested volunteers who are part of a community of practice and work on offline solutions in critical situations. She performed empirical ethnographic work on-site at two animal evacuation locations to understand the information management needs of volunteers helping animals in disasters.

Second Question: Analysis and Queries

Based on her interest in ethnographic study, Analyst 1 works on different types of social data such as Facebook and Twitter data. She needs to gain understanding of the flows of information, the type and format (video, images, text) of the information, who is producing and engaging with it, how it is being organized and presented, and what activity is happening around it. She is always interested to know who the most active users and the most influential are (sometimes relying on
Klout score) to track their activity and connections, and potentially for following up with them for interviews once the disaster is over. Analyst 1 investigates tweets manually looking for entities like landmarks, roads, town names, etc. She has used Hootsuite to help her track hashtags and users in real time. Over the time of the disaster, Analyst 1 tracks the growth of a user’s followers and the patterns of that growth. Sometimes, she needs to revisit the data afterwards, particularly, if she sees spikes in the number of people sharing interesting information or a new development in the disaster event that brings focus to certain parts of her research.

Third Question: Impact of Real-Time Analysis

Analyst 1 believes that real-time analysis would add greatly aid her research practice. Real-time analytics can enrich crisis informatics research with a quick understanding about a disaster and open different ways to do further studies on the event. Analyst 1 would like to be able to track in real time trending hashtags found during an event. As she is interested in the content of tweets and the information they carry, she wants to instantly know who tweeted certain information, not necessarily looking for the most active users. Normally, Analyst 1 prefers to monitor a disaster, then contact interesting users near the end, and request an interview. However, if there were something in particular that she needs to know right then and there, this would allow her to react quickly contacting those active users immediately to collect more individualized information. Moreover, extracting entities like landmarks, roads, towns names in real time would provide her with contextual understanding, particularly for informal or colloquial references to these entities.

Fourth Question: Views and Output

Analyst 1 likes to view real-time social data in different ways. She would visualize all trends in data related to an event. She would like to track, for instance, the top ten Twitter users or the top ten most retweeted tweets every hour with many options for persistence. She would prefer to be able to use different types of charts such as line graphs, bar graphs, histograms, and pie charts.

Fifth Question: Event-based Decision Making

Analyst 1 thinks support for event-based decision making would be very valuable. A notification sent to her mobile device alerting her of unusually high activity around a keyword that she’d pre-identified would enable her to get a collection started as soon as an event starts. She also thought it would be useful if a collection could automatically begin when a spike on certain
words/locations occurred. Analyst 1 stressed the need to be sure that the analysis environment is capturing ‘enough’ of the Twitter stream in real time. Many questions need to be answered here: Are we getting all instances of tweets with our specified keywords? What are we missing? Are we at saturation? Is the data noisy? Who are the key users here, and what hashtags are they using? She added that if the system could identify the top Twitter users around a keyword every 5 minutes from the point of the first spike in data, that might help narrow down on the interesting ones. Also, if the system could capture ‘snapshots’—graphs of data volume, active users, etc., at time-spaced segments across the disaster, and automatically store it in an ‘Event Tracking document’ (a template that analysts had to fill in manually)—that would greatly aid the analysts during the event and support research paper writing after the event. This would allow them to track the event over time, not only in real time.

Sixth Question: Benefits of Real-Time Analysis

Analyst 1 sees many great benefits for doing her research in real time. A real-time analytics tool would help her perform her inductive analysis as the disaster unfolds, informing decision making on additional data to collect and on potential interesting research questions. As she turns to use some available online tools like Hootsuite to track tweets in real time, she appreciates having more customized tools to collect and organize data that is relevant to her topic of interest as soon as it gets posted and shared. She would track popular related hashtags and influential users faster then follow up with deeper analysis as early as possible. Most of the time, Analyst 1 needs to backtrack data based on time, by hours, days, or weeks, manually which is currently labor intensive. Leveraging real-time analysis, would help her track social data posted in different stages of interest of a disaster instead of waiting for the event to be complete and then to backtrack all of the data she needs. She says: “An example to help you understand this: In the earthquake/landslide just before I left Project EPIC, I had to manually take screen grabs and downloads of data on Facebook and in particular, of spreadsheets identifying missing people, and of public maps of where people were missing from, so I could track the changes over time. It was time-consuming and I’m sure I missed some key change times, where realizations were made and data adjusted. I was not able to gather the stream of data, and not able to grab it as it happened. I’d have loved a tool that would automatically identify that activity was happening, and begin grabbing screenshots or downloads of those changes across periods, which would help me knit together the narrative of activity. Now you're getting into something very useful”
Seventh Question: Challenges with Real-Time Analysis

Analyst 1 finds some challenges doing real-time analysis. She would like the proposed platform to support the analyst activity, not be something that pushes the analyst into a structure. It therefore needs to cross and combine different social channels, and use time as its basis. Another area of challenges is filtering decisions. Analysts do not want to grab every single thing, but make decisions about what to grab and where to grab it from is hard. Analyst 1 often goes with her gut in making decisions, and has “lucked out” by watching Facebook Page growth, for example, to help her identify which page to watch and focus her data collection on. She believes automated monitoring of Facebook page growth would thus be useful.

Analyst 2

First Question: Current Research Practice

Analyst 2 has been involved as a volunteer collaborating with emergency response organizations providing/studying social media monitoring support during different events like the 2013 Colorado Flash Floods. As a crisis researcher, her research practice is devoted to supporting emergency responders with social data disseminated by members of the public during an event. Leveraging social media, Analyst 2 looks for data produced only by individual voice Twitter users (as opposed to news organizations) with the aim of monitoring those threads of social data and finding significant content to be reviewed by official responders. She also aggregates data from multiple incidents to support her research.

Second Question: Analysis and Queries

Analyst 2 works with large sets of tweets looking for users who contribute useful information for emergency response, something she calls “individual voice” tweets. Hence, she needs to filter out all tweets from official responders, spam accounts, and known sources of mass media, to investigate tweets only posted by the remaining individuals. She then explores the filtered set deeply looking for more meanings and categorization of data in terms of emergency situations and reportings. Analyst 2 relies on software tools like Google OpenRefine and Tableau to explore, clean, and transform large data sets of tweets. For easier data filtering and refinement, she annotates data based on specified categories then visualizes the annotated data by different ways. She also writes and runs some Python code to conduct deep data analysis and classifying.
Third Question: Impact of Real-Time Analysis

Analyst 2 simulates getting data in real time by collecting social data regularly during an event and then analyzing it quickly. She tries to be up-to-date with the latest streams to work on urgent data as soon as possible. Analyst 2 appreciates working in real time and looks forward to moving all her analytical questions to the first stages of disasters. She would track more information posted to collect all interesting data and investigate interesting user accounts more specifically.

Fourth Question: Views and Output

Since many filtering and annotation phases need to be performed, Analyst 2 would like to view every step and visualize it for better explanation and conveyance. She needs to try different types of graphs and charts to view results in real time, as she used to do using Tableau with her collected data sets. She works with CSV files as a primary format, but she’s interested in working with multiple formats for more flexibility.

Fifth Question: Event-based Decision Making

Analyst 2 provided good insights on how real-time event-based decision making could help her. She would like to be notified if there are any sudden spikes of incoming tweets happening while the data is being collected for a specific disaster. This will guide her to make further investigations, whether to collect more data on new keywords or ask questions by running some queries to reveal some interesting facts about the current spike. Analyst 2 would like also to track a list of keywords of interest for her. Real-time analysis will allow her to be updated with what is going on around some keywords, how people are spreading those keywords, or how they are converging around them.

Sixth Question: Benefits of Real-Time Analysis

Analyst 2 believes going real time with her research would have huge advantages. She would move all her practices to real time when a disaster strikes and emergency response is more critical. This gives her more opportunities to follow interesting users and to look for related data. She would track her filtered and annotated data sets and investigate them as they accumulate over time.

Seventh Question: Challenges with Real-Time Analysis

Analyst 2 envisioned some challenges with real-time analysis. Drawing from her experience with emergency social media monitoring, Analyst 2 finds it is difficult to keep up with high
volumes of tweets in real-time, as there are often spikes of data related to certain events. So she thought it would be useful to be able to segment incoming data based on keyword or sets of keywords. She also referred to TweetDeck as an incredibly useful interface that helps her to tame volume issues by allowing her to view multiple timelines at the same time. As Analyst 2 believes that real-time analysis is great, she recommends that there should be some way to grab information as it scrolls by and organize it. Also performance could be an issue, especially in large events. It might be useful to be able to bound the data in different ways for queries e.g. only today’s information, or within the last hour, etc.

Analyst 3

First Question: Current Research Practice

Analyst 3 is interested in studying online socio-behavioral phenomena during disasters. She studies how information is diffused across a population, what different retweet patterns exist, and how this information differs when it spreads from people geographically located in the disaster to the general public. She is also interested in studying the online social relationships of people and how social networks emerge in relation to a disaster. She tracks all social activities that happen such as when users follow others, or when people retweet or reply to tweets. She explores all kinds of online cooperation to identify the types and structures of social interactions. Many questions need to be answered here in terms of how, when, and why the public interact using social microblogging during crisis. Analyst 3 needs to look at different data sets by breaking up large data sets into smaller and more manageable sets. She uses a social network tool called “NetworkX” to help her explore and view all Twitter users and their relationships and activities.

Second Question: Analysis and Queries

Analyst 3 needs to filter huge data sets by many criteria such as by users, by locality, by date range, or by language. She also needs to create and analyze different data sets separately based on retweet behaviors or contextual streams for users of interest. Most of the time, she needs to collect further information about the data set such as the followers for an interesting user. Analyst 3 wishes to make working with huge number of tweets easier, because she would need to ask more and more questions as she goes deeper into her social analysis.
Third Question: Impact of Real-Time Analysis

Analyst 3 believes working with social data in real time would be beneficial to her research. She finds tracking influential users, for instance, while an event is fresh very rewarding instead of waiting for the event to end. This allows her to start analyzing data right away and tracking all social activities as the time goes on. She is also interested in chunking data streams into fixed time slices based on minutes, hours, or even days. However, she would like to decide on the time slice based on the activities she observes while data is streaming in, to know the right chunk for each disaster event. Analyst 3 would like to incorporate some machine learning techniques and methods on large sets of tweets. Classifying tweets based on different languages is an example. Moreover, topic modeling techniques would add more depth to her research questions. She is also thinking of looking for indications of trust within the contents of the data in future work.

Fourth Question: Views and Output

Analyst 3 would like to explore the distribution of how many user connections are found in specific data sets, or how many tweets are being retweeted during an event. She also needs to explore different kinds of social networks found in her data sets and to traverse those networks either interactively or via queries. Analyst 3 wants to deal with data in various ways. She would need to use JSON or CSV. Most important for her is to get data that can be imported into social networking tools like NetworkX, and to be able to analyze the data using her own python code.

Fifth Question: Event-based Decision Making

Analyst 3 has experienced the process of monitoring keywords and search terms for different disaster events. She thinks a tool that allows her to track the trends of words over time would be very helpful to add additional keywords to EPIC Collect as the event unfolds. This will enrich the collected data sets and include more relevant tweets. This feature would also allow her to turn off keywords that have become inactive. She also refers to the daunting task of tracking co-occurrence of keywords manually that helps her find more relevant words, used for more data streams to be collected. Analyst 3 worked on the data collection of the “2014 Ukrainian Protests” in which 151 search terms are used in different languages. However, if she relies on a decision making tool that runs and tracks interesting words based, for instance, on predetermined thresholds, this would be very helpful and could save her a lot of time.
Sixth Question: Benefits of Real-Time Analysis

Analyst 3 believes doing social analysis in real time would be rewarding and beneficial. She finds it very important to start looking at the streaming data and ask different questions upon it while the event is active. It will speed up monitoring of important terms and increase the possibilities to have a comprehensive data set that covers all data and social activities happening within the event. It would save her time from having to go back to conduct further investigation or additional data collection. Analyst 3 also believes that the ability to have different types of social networks, such as the retweet network, followers network, replies network, mentions network, etc., and the ability to explore these networks in real time would be helpful for her research.

Seventh Question: Challenges with Real-Time Analysis

Analyst 3 finds real-time very challenging when collecting large sets of social network data (i.e. graph-based data). The growth of these networks over time can be quadratic or even exponential even for modestly sized crisis events. The different types of social networks that she explores are huge and complex. Working with such large, complicated data makes it difficult to manage and get data quickly. Analyst 3 also thinks the user interface of my proposed analytic tool needs to be flexible to be useful.

Analyst 4

First Question: Current Research Practice

Analyst 4 is interested in studying the collaborative work of crisis mapping using the OpenStreetMap platform. Analyst 4 focuses on geo-referenced social data disseminated during disasters in social media like Twitter to learn about all kinds of social behaviors. Geo-tagging is a crucial piece of information for Analyst 4 in order to identify locations of sources. He is interested in learning about the movement patterns of people affected by disasters, and comparing the decisions they make with the information they share during mass emergencies. Studying this difference can, for instance, help him to identify interesting evacuation patterns during mass emergencies (such as users who clearly evacuated based on their actions but never once said they were evacuating online). Overall, Analyst 4 is interested in applying a wide range of social analysis techniques to disaster events.
Second Question: Analysis and Queries

Analyst 4 needs to apply different kinds of filters iteratively dividing a huge underlying dataset into smaller manageable ones that contain only pertinent data of interest. As looking only for georeferenced Twitter data, Analyst 4 first filters out all tweet objects that are not geotagged. Then he classifies the data based on different sources of geotagging information found in the data, such as using Instagram or foursquare to verify locations found in tweets. Further filtering can then be done based on different time periods. Analyst 4 uses some “geo-aware” software in his research; such as using Leaflet to visualize his geotagged datasets. He also writes code using JavaScript and ruby to perform custom analysis on his datasets.

Third Question: Impact of Real-Time Analysis

Analyst 4 is interested in doing real-time analysis for his current research practice. He would ask many questions in real-time to explore incoming data from social media. He could discard data with no geotagging information right away upon collecting any data of interest. Generally, Analyst 4 would like to track all current peaks in the streaming data. Knowing the peaks in hashtags, mentions, URLs, etc., in real time would allow him to quickly understand important facts about an event. He would also like to apply some topic modeling analysis techniques to reveal what the most important topics are in an event. This helps him learn about the types of information the public post during different stages of disasters, for instance, whether this information is related to awareness and alerts or related to assistance. Analyst 4 would also like to use some social network analysis to explore, for instance, how top Twitter mentions change during time.

Fourth Question: Views and Output

Analyst 4 would like to work with multiple data formats for more flexibility. He needs to use geoJSON data, since he uses a lot of geo-aware APIs. He also wants to be able to select specific attributes of tweets instead of having to deal with large tweet objects. Map visualization is also a very important tool for Analyst 4’s work, so that he can interact directly with geotagged data.

Fifth Question: Event-based Decision Making

Analyst 4 showed some interest in applying real-time decisions triggered by exceptional situations. According to his area of interest in tracking geotagged social activities, Analyst 4 would like to be notified if an area within the geographical boundary of a disaster is showing very high
activities in information diffusion. Similarly, very low activity is also critical and would inspire him to do more investigations on such areas. Moreover, with respect to crisis mapping, he would like to be notified if crisis mappers are converging in some areas and, likewise, if they had diverged in other areas. Such situations are worth tracking and understanding. He suggests to use an online dashboard to track real-time notifications and alerts, and flexible interfaces to create different kinds of event-based real-time decisions that are capable of plugging in user-generated code.

_Sixth Question: Benefits of Real-Time Analysis_

Analyst 4 believes real-time analysis can bring value to his current research. He believes that going real-time in social media analysis can identify important emerging patterns of social behavior in crisis. Real-time analysis can add many opportunities to do deeper analysis and focus on different areas of interest at the very early stages of a disaster. Tracking influential Twitter users in real-time, for instance, is important to understand current and evolving data diffusion patterns. Moreover, social network analysis when incorporating real-time data could reveal different types of social activities and lead analysts to elicit more interesting data than they would with post hoc analysis. Applying real-time analysis in crisis informatics research helps analysts clearly understand the type of information shared by the public during different stages of a disaster.

_Seventh Question: Challenges with Real-Time Analysis_

Analyst 4 is a crisis informatics researcher and a software developer as well. He sees many challenges with respect to incorporating real-time analysis into the work conducted by Project EPIC. The challenges that Project EPIC faced creating the current set of data management systems (EPIC Collect and EPIC Analyze) were daunting and he can see that doing the same thing in real time would bring entirely new challenges. Analyst 4 believes that real-time big data analysis requires high performance, fast and iterative results, and more user interactions with the data to get the analysis right.

_Analyst 5_

_First Question: Current Research Practice_

Analyst 5 is interested in studying information diffusion in social media during disasters. This research direction requires much effort with respect to exploring, detecting and applying different modeling techniques in order to understand what, why, and how information spreads during and
after a disaster. Analyst 5 studies all kinds of information diffusion including textual, graph-based, and media-based data. She tracks all kinds of information that the public posts and shares, looking for the value this information provides in support of disaster awareness, response, or recovery.

**Second Question: Analysis and Queries**

Analyst 5 needs to apply many filtering and exploration techniques to understand the social media data collected for a disaster under study. She looks at many metrics that help her explore an underlying dataset, such as the most influential users or the most retweeted tweets. She filters data tweeted in disasters based on local users versus non-local users to reveal any differences in information diffusion. She also works on data sets containing URLs or embedded media such as pictures or videos. This allows her to analyze the patterns of image diffusion and categorize image contents based on whether images convey collapsed buildings, missing people, or other important information. Most of the time, data sets are large even when applying many filtration steps. Analyst 5 has worked in Tableau by importing data sets containing up to 1.5 million tweet objects and then applying many filters and visualization methods for further data exploration. Using Tableau, Analyst 5 has experienced some degradation in time response when working with large datasets.

**Third Question: Impact of Real-Time Analysis**

Analyst 5 is interested in asking many questions in real time upon the streaming data when a disaster has just started. She would like to track trending topics over time, looking for emerging topics in the different stages of a disaster. She finds it is important to distinguish during the first stages of disasters between local Twitter users and those who are collaborating from a distance. Tracking data in real time will help her find patterns in the types of tweets being generated and the content within them across different types of disasters.

**Fourth Question: Views and Output**

Analyst 5 keeps track of different datasets of tweets and saves them for further investigation and visualization. However, leveraging a real-time analytics tool, Analyst 5 would like to be able to dynamically generate different types of graphs and charts as she discovers interesting incoming data. With real time capabilities, Analyst 5 would like to track tweeting and retweeting activities of pictures posted during an event. She also looks for more variety in data formats, specifying that JSON and CSV files would be helpful.
Fifth Question: Event-based Decision Making

Analyst 5 thought of some scenarios that could allow her to make important analytical decisions using a real-time analysis tool. She would make decisions on triggering information such as significant numbers of retweeting activities, image distributions, or topic dissemination. Analyst 5 has also participated in the process of collecting keywords and search terms to be used for data collection of tweets during events. She believes if she could find a way to speed up this process, this would be great, instead of searching for relevant terms by manually exploring all streaming tweets and tracking candidate terms.

Sixth Question: Benefits of Real-Time Analysis

Analyst 5 believes that doing real-time analysis will add many capabilities to her current research. She will be able to start studying information diffusion much earlier and answer questions like “what is happening right now?” She would explore data that could be lost if it was not discovered in real time. Looking at data in real time would give her better ideas of what the incoming data looks like and help her decide what kind of data needs to be collected and analyzed.

Seventh Question: Challenges with Real-Time Analysis

Analyst 5 sees a big challenge in doing real-time analysis in her research work. As focusing on data diffusion during disasters, she would find it difficult to track a huge amount of incoming data, and know what and why to ask. Thus, she would rely on any provided visualization capabilities to look for interesting patterns or meanings of information diffusion revealed from the incoming data.

Analyst 6

First Question: Current Research Practice

Analyst 6 is a crisis informatics researcher who is interested in studying the crowd work of mapping activities in crisis. He is not directly working with Twitter data, however, I found his insights important to the contribution of my thesis in terms of real time analysis of social data and social collaboration during disaster events. Analyst 6’s research work is devoted to issues surrounding open map data and, in particular, to the use of OpenStreetMap (OSM) during crisis. As a founding member of the Humanitarian OpenStreetMap Team, he is currently interested in studying the socio-technical evolution of OSM in response to humanitarian events of disasters,
with many goals of analyzing the global growth of crisis-related OSM communities and collaborative spatial data production among distributed volunteers.

**Second Question: Analysis and Queries**

Analyst 6 studies many OSM-related social activities and behaviors. He analyzes social interactions and cooperations of globally distributed volunteers who participate in mapping activities during a disaster event. He needs to examine OSM data posted during an event of interest. Thus, many processes of data importing, filtering, and querying are required to explore and analyze different mapping activities.

**Third Question: Impact of Real-Time Analysis**

Analyst 6 believes analyzing social data in real time is beneficial. First, crisis researchers can quickly understand what main things are happening. From his point of study, many questions can be answered like: Who are the active mappers? What are the priorities of mapping activities? Are these activities about mapping roads, buildings, shelters, etc.? and Where are these most mapped things located? What problems or difficulties are those volunteer mappers facing? All of these questions asked in real time will assist to know what kind of data is required to collect for further analysis. Second, incorporating a live dashboard, showing all real-time active mappers and their mapping activities, helps the OSM community to understand their own behaviors and creates more motivations to collaborate on more geographical areas. A dashboard can show important live metrics like how many streets or how many miles have been mapped since a disaster has started.

**Fourth Question: Views and Output**

As an OSM researcher, the most interesting view Analyst 6 would like to see is real-time mapping in an area of interest. This would reveal what kinds of mapping activities are taking place. Analyst 6 would like to deal with many different graphs, such as a chart of most active users along with the number of map edits, a chart of all mapping activities over time, etc.

**Fifth Question: Event-based Decision Making**

Analyst 6 had some insights on how he could leverage real-time analysis capabilities to make important analytical decisions while crisis events are still active. As the crisis unfolds, Analyst 6 would make decisions based on incoming data generated by active mappers. He would like to be notified about interesting mapping activities happening around certain areas or performed
intensely by some active mappers. He could follow particular users or organizations to look for some interesting behaviors. Furthermore, being notified of the currently active mappers, Analyst 6 would like to be able to contact those active volunteers to obtain more information that can only be obtained during the crisis, aiming for collecting their ongoing experiences of the event.

**Sixth Question: Benefits of Real-Time Analysis**

Analyst 6 believes that analyzing massive amount of crisis-related social data in real time will add many features to crisis informatics research. As an OSM researcher’s point of view, tracking data in real time allows the OSM community to understand what is happening now, what is interesting about the current activities, and what types of data need to be collected and tracked, so analysts can reveal new meanings and hidden patterns in social collaboration during crises.

**Seventh Question: Challenges with Real-Time Analysis**

Analyst 6 finds real-time analytics is challenging. As for software developers, the data is large, unwieldy, and hard to process. Every query is computationally expensive. As for analysts, querying the underlying data streams is challenging. Thus, the analytical platform would need to support very flexible and easy to use querying tools.

**Analyst 7**

Analyst 7 is a leading researcher in human-computer interaction, computer-supported cooperative work, and social computing. I conducted a quick interview with Analyst 7 to obtain a higher level of insight to my study. The first question I asked concerned the current research practice of Project EPIC. Analyst 7 gave an overview of the work that has been recently done and pointed out some areas of interests as different research directions in the lab. One of these research directions is to study and reveal highly localized social data disseminated in disasters and how this could deliver important knowledge to the public in general and to emergency responders. Another track is how to collaborate with emergency responders and provide them with insight to the vital information posted in social media by the public during disasters, and not by official sources, mass media, or online news. Another important track is to study the information diffusion during different stages of disasters and how geolocations and time intervals of a disaster could affect the type of this information.
The second question was about how real-time analysis could augment the current research practices. Analyst 7 affirmed that moving social data analysis from after the event to real time would provide huge advantages to Project EPIC. She asserted that real-time data analytics will give a better sense of what is going on straightaway and add instant access to navigate the digital space during a crisis event. Crisis informatics researchers will have many different research questions. Analyst 7 believes that they missed many opportunities in the past to ask questions in real time, and they were limited by the capabilities of doing social computing after the event is over. If they would have been able to get real time exploration they would have dug deeper in the data looking for more related information. Analyst 7 also pointed out to the importance of leveraging some event-based decision making that would help them add important notifications or alerts. The most critical example she mentioned is the need to add more capabilities to the process of collecting data for crisis events. Analysts need more hands-on tools that can help them find more relevant search terms to be used in their data collection. Analyst 7 asserted that relying on common words like “Boulder Flood” or “Japan Earthquake” are not enough to cover all the tweets that talk about the disaster. They need to look at co-occurrence of terms posted and shared by locals, for example, to include all mentions of places or areas names to the overall data collection. Here is where analysts can apply linguistic algorithms to support them in making decisions. At the end, Analyst 7 referred to the importance of having real time mapping of tweets to show the relative areas of online activity for an event.

3.2.2 General Findings

My interviews with analysts have provided insight into their current and future research requirements and provided a clear picture of the potential benefits for real-time big data analysis. Table 2 below summarizes the interviews results. The interviews shed light on each analyst’s research interest and direction with respect to analyzing crisis data. These directions include studying and analyzing the socio-technical cooperative work of the crowd, the types and evolution of online social relationships and social interactions, the collaboration of emergency responders and the public, the types of information and media disseminated and shared during different stages of disasters, and the significant differences of data posted by locals versus nonlocals.

Although some of them do off-line social studies that require personal efforts for collecting further information or conducting ethnographic studies, all analysts deal with online social data in
some way and need different types of data exploring and filtering techniques. The first step to explore an event is to acquire some general metrics about it such as the date range of the underlying data set, the number of tweet objects, the number of unique users that contributed to the data set, the keyword list used for collecting the data set, etc. Then, analysts start brainstorming to ask many questions about the event under study, run (currently) as batch-oriented tasks. The most important and common questions they ask are those for identifying the most active users, the most influential users, the most retweeted tweets, and so on. Most of the times, analysts need to reduce the size of high volume datasets by dividing them into smaller sets in which they would be able to apply and manage queries and analytical studies. Usually, the dividing process is done based on the stages of disasters such as warning, evacuation, and recovery. Some other criteria could be based on geo-referenced data, local versus non-local contributors, or manually-annotated data.

<table>
<thead>
<tr>
<th>Question Type</th>
<th>Summary of Answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Research Practice</td>
<td>- Socio-technical cooperative work of the crowd.</td>
</tr>
<tr>
<td></td>
<td>- Types and evolution of online social relationships and social interactions.</td>
</tr>
<tr>
<td></td>
<td>- Collaboration of emergency responders with the public.</td>
</tr>
<tr>
<td></td>
<td>- Information diffusion and media disseminated and shared during different stages of disasters.</td>
</tr>
<tr>
<td></td>
<td>- Differences in data posted by local people versus non-local remote collaborators.</td>
</tr>
<tr>
<td>Analysis and Queries</td>
<td>- Get general metrics about the event such as number of tweet objects, number of unique users, date range, search terms, etc.</td>
</tr>
<tr>
<td></td>
<td>- Identify the most active users, most influential users, the most retweeted tweets, and so on.</td>
</tr>
<tr>
<td></td>
<td>- Reduce the size of large datasets into smaller manageable sets.</td>
</tr>
<tr>
<td></td>
<td>- Filter data based on geo-reference, local versus non-local contributors, or manually-annotated data.</td>
</tr>
<tr>
<td>Impact of Real-Time Analysis</td>
<td>- Get live metrics for active crisis events.</td>
</tr>
<tr>
<td></td>
<td>- Track the most influential users during time.</td>
</tr>
<tr>
<td></td>
<td>- Identify users’ activities and their interactions.</td>
</tr>
<tr>
<td></td>
<td>- Focus on data posted by local individuals.</td>
</tr>
<tr>
<td></td>
<td>- Track only media-attached tweet data.</td>
</tr>
<tr>
<td></td>
<td>- Use machine learning methods, such as real-time classifier algorithms.</td>
</tr>
<tr>
<td></td>
<td>- Add topic modeling techniques.</td>
</tr>
</tbody>
</table>
### Real-time Views and Output

- Visualize results in different ways, such as line graphs, bar graphs, histograms, or pie charts.
- Incorporate social graphs.
- Use multiple formats, such as JSON, geoJSON or CSV files.
- Allow data sets to plugged into many different analysis environments.
- Write and run some python code on data sets.

### Event-based Decisions

- Decide on new keywords or search terms to collect more relevant data.
- Track sudden spikes in tweet volume.
- Track high activity around keywords.
- Get notified on co-occurrence of the names of places or areas posted and shared by locals.
- Track high or low activities in some areas within the geographical boundary of a disaster.

### Benefits of Real-Time Analysis

- Ask questions while a disaster event is still alive.
- Ask questions as they explore the incoming data.
- Track the trend of topics as they happen.
- Capture additional data as they come in.
- Start off-line investigation from the first few days of disasters.
- Track interesting social interactions and capture them immediately.
- Explore convergence in social media in different stages of disasters.

### Challenges with Real-Time Analysis

- Doing real-time analysis is a different type of challenges.
- Requires high performance and fast results.
- Analysts should know what and why to ask in real time.
- Requires user interactions iteratively with results.
- Requires flexible and easy to use user interface.

### Table 2: Interviews Results

With respect to leveraging a real-time analytics tool, all of the interviewed analysts were interested in moving their research practices to real time. Most of the analysts have stated how they would ask questions right away when a disaster strikes. They would explore the current social data being posted and shared in social media and track how conversations are evolving. Furthermore, all analysts confirmed the continued need for batch-oriented analysis even with real-time analysis capabilities. In addition, a dashboard can show important live metrics for active crisis events. The interviews show that every analyst has different interests with respect to social data streams. Some of them would like to track the most influential users over time, some would track
users activities and their interactions, other would focus on data posted by local victims or local responders, and some would like to track only posted images. As for more data analysis techniques, some analysts showed interest in incorporating some machine learning methods on large sets of tweets, such as real-time classifier algorithms as well as applying topic modeling techniques for more in-depth data analysis.

Analysts would like to be able to track the information they are interested in and have capabilities of visualizing results in different ways, such as line graphs, bar graphs, histograms, and pie charts. Also, incorporating social graphs is important to visualizing large and small social networks. Analysts find that it would be flexible if they can download data sets in multiple formats, such as JSON, geoJSON, or CSV files. More importantly for some analysts is that data sets can be exported for use in another analysis environment. Finally, some analysts would like to be able to write their own code and apply it to the data streams and historical data sets.

Furthermore, analysts saw the need for event-based decision making capabilities to allow them to capture data of interest in some way. Some of the analysts had good insights on how they could get notified of new keywords or search terms that allow them to collect more relevant data and make sure they do not miss any crisis-related information posted during an event. For example, analysts would like to be notified of the co-occurrence of terms posted and shared by locals who could mention many names of places or areas that were affected by a disaster. With event-based decision making, analysts would like to track spikes in tweet volume or high activity around keywords. An analyst would like to be notified of both high or low activities in information diffusion in some areas within the geographical boundary of a disaster. Analysts would like to use an online dashboard to track notifications or potentially through their mobile devices. They realized that applying rule-based event-based decisions could guide them to do more investigation and reveal important hidden knowledge in crisis data.

All analysts believe that leveraging real-time analysis features in Project EPIC will add many capabilities to their current research practices. They find it is very important to start looking at social media streaming data and ask different questions upon it while a disaster event is still active. Analysts realize the benefits of moving their research activities from after the event to during the event. This will allow them to ask questions as they explore the incoming data. For instance, they can track the trending topics as they happen, enabling them to capture additional data as they come
in. They can start off any required off-line investigation from the first few days of disasters instead of waiting until an event is over. Analysts also find tracking social activities in real time will guide them to follow all interesting interactions and capture them immediately. Analysts also emphasize the advantages of generating different datasets across different time periods, in such a way they could explore important convergence in social media in different stages of disasters, from preparing and warning stages to recovery stages.

However, most of the analysts find it challenging to incorporate real time analysis with their current tools. Real time requires high performance, fast results, and more user interactions with the dataset and the results of queries. Another type of challenge in doing real-time analysis is filtering decisions. In order to interact with real-time data streams, analysts would need to know what to ask and why, so they could reveal important meanings from data, and eventually get the most benefits of analyzing data in real time. Analysts need to work with iterative results and more interactions with data. As a result, they would like to have a flexible and easy to use user interface to submit queries. Visualizations would also help them to look for interesting patterns or meanings. Another recommendation is to partition incoming data based on, for example, days, or any time periods, for easier tracking of high volume data. With this summary, I present the requirements for my platform next.

### 3.3 User Requirements

My investigative study can be used to elicit requirements for a comprehensive data analysis environment based on the Lambda Architecture initially targeting the domain of crisis informatics but eventually supporting a broad range of application domains. Earlier, I made the assertion that solving this research goal for crisis informatics would produce a system that would have benefits for other application domains; I believe the wide range of interests, tools, and practices of my target set of analysts already represents a diverse set of use cases and problem domains, lending support to my assertion. My study has helped me gain a deeper understanding of the needs of data analysts and this has highlighted some challenges that need to be addressed.

**Challenge 1:** Analysts need to work on historical datasets along with streaming data. An example is when analysts are still interested in a past event, or working on an active event but this
event spans a period of several months such that some part of the event’s dataset has already been migrated and stored as a historical dataset.

**Challenge 2:** Analysts need flexibility to collect further data at any point of their analysis tasks. One example here is collecting user contextual streams for a unique user to retrieve a context for a particular tweet. Another example is collecting a followers list of a particular user of interest.

**Challenge 3:** Analysts need a flexible analysis tool to help them decide what questions they should ask in real time on a data stream. When a crisis event unfolds, my analysts need to react quickly to incoming data and not spend time fighting with their analysis tool.

**Challenge 4:** Analysts face different kinds of crisis events and mass emergency situations, such as natural hazard disasters like earthquakes, hurricanes, wildfires, and floods or human-generated disasters like protests, ethnic violence, terrorism or industrial accidents. Each event is different in terms of its volume, impact, or response, and this brings different challenges to the analysts. This means that analysts need to think differently for each event, and their analysis tool needs to be flexible enough to support them.

I tackle all these challenges when designing EPIC Real-Time. Next, I discuss my proposed design for queries that considers most of the requirements and challenges discussed above.

### 3.4 Types of Queries

Analysts need to ask ad hoc queries on their datasets of interest and to address the third and fourth analytical challenges above. I designed a wide range of query types to handle the broad range of questions that analysts ask. Using the information gleaned from my interviews, I identified three types of queries: dataset, trend, and popular. These are discussed in detail below.

#### 3.4.1 Dataset Query

The dataset query calculates the common metrics of interest for all crisis datasets consisting of Twitter data. This query is executed by default for all events; however, an analyst can request different time periods for each metric if needed. The dataset metrics include:

- List of unique users.
- List of unique hashtags.
- List of unique URLs.
- List of unique mentions.
- List of unique languages.
- List of unique locations.

### 3.4.2 Trend Query

The trend query is an aggregate query that identifies all kinds of trends found in a dataset. By default, it calculates the following trends.

- Trending hashtags.
- Active users.
- Most frequently posted URLs.
- Most frequently posted user mentions.

This query takes five parameters: lower and upper time bounds, a threshold limit or top k, and the tweet attribute for which the trend is calculated. The output is a list of values and counts of each trend found for the selected tweet attribute. The lower and upper time bounds define a time period for the requested trend (e.g. from October 1st to October 10th, the last three days, or the first five days of an event). If the lower and upper time bounds values are not specified, this will indicate that the query is a real-time query that works only upon Twitter streaming data. A threshold, specified by the user, is the minimum number that a value should appear in the data collection of streams to be considered trending.

An example of a trend query would be to query for a trend in hashtags for an event. An analyst would specify the hashtag attribute, select a time period, and decide on a threshold limit that specifies how many times a hashtag must occur within the time period to be considered a trend. Then, the query returns all trending hashtags along with the number of times they occur. If a threshold limit is not specified, the query will generate by default the top k hashtags found in the dataset. A second example would be to discover the most active users for an event. An analyst would specify the user attribute, select a time period, and decide on a threshold limit that specifies the number of times a user must have tweeted to be identified as an active user. Then, the query returns all active users along with their activity.
3.4.3 Popular Query

The popular query is a type of query that identifies information dissemination within a dataset; it identifies data popularity in a dataset by providing the following two types of diffusion:

- Popular tweets (most retweeted tweets).
- Popular users (most retweeted users).

There are also five parameters for this query: the lower and upper time bounds, a threshold limit or top k value, and an attribute that specifies the type of popularity. The output is a list of frequencies of the popular values found. For popular tweets, the query returns a list of tweets along with how many times each is retweeted within a given time period. Similar, for popular users, the query returns a list of users along with how many times each is retweeted within a given time period. A threshold specified by the user is the minimum number that a value of a tweet or user should be retweeted in the data collection of streams to considers popular. Again, the lower and upper time bounds define any range of dates, and no values of these two time bounds creates a real-time query. For example, for getting a list of popular tweets, the analyst selects a time period and decides on a threshold limit that specifies the number of times a tweet must have been tweeted to be identified as a popular one.

3.5 System Overview

The investigative study with the analysts revealed their requirements with respect to performing crisis informatics research in real time. My goal is to support these analysts with their journey to *wisdom* as shown in the DIKW hierarchy (see Figure 9; Wikipedia, DIKW), a standardized representation in information science showing the relationships between data, information, knowledge, and wisdom. In performing crisis informatics research, the analysts will move from *data* to *information* to *knowledge*, and eventually to wisdom about what occurred in that event with respect to their research questions. Typically, raw data is processed to form meaningful information. This information is analyzed to generate knowledge. Finally, the creation of knowledge leads to wisdom. While knowledge is valuable to share, wisdom is a state that combines prior experience, interpretations, and wise judgements, to new situations, exactly what the analysts need to perform research in crisis informatics. So, I endeavor to improve the tasks and operations
these analysts can use to create knowledge and to acquire wisdom—defined as the ultimate level of understanding and the most favorable outcome (Bellinger, et al., 2004).

![DIKW Pyramid](image)

**Figure 9: DIKW Pyramid**

To achieve this, I designed a system called EPIC Real-Time. This system transforms Project EPIC’s batch processing approach to analyzing data to one that can also process data in real time. Figure 10 gives an overview of these two contexts and my goal state: batch processing, real-time processing, and combined. In batch processing, one collects data streams and stores them into a data store, then at a later time, one extracts and processes data to be available for interactive analysis. In real-time processing, one collects data streams and processes them immediately to get them available for real-time interactive analysis. In the combined context, one blends the two types of processing to get the capabilities of both.

My new platform EPIC Real-Time is fundamentally oriented around a stream of high-velocity events; initially my events are tweets from Twitter but eventually could be more heterogeneous and include Facebook posts, OpenStreetMap edits, pictures from Instagram, etc. These events are processed by a scalable stream processing engine that simultaneously queries and integrates data to discover trends that are presented to analysts via live dashboards. EPIC Real-Time is meant to provide a complete set of analysis features applied on both streaming and historical data while providing flexible queries, fast response time, reliable performance, and high availability.
Figure 10: Three Different Big Data Processing Contexts

The three challenges I face in meeting the research goals of this thesis are: 1) Enabling real-time big data analytics, 2) providing access to analysts, and 3) engineering a lightweight scalable big data platform. I discuss these challenges below.

3.5.1 Enabling Real-Time Big Data Analytics

EPIC Real-Time needs to provide real-time analytics on streaming Twitter data. The power to extract meanings and facts from this data should not come at the expense of speed. No matter how large the data set is, data insights need to be generated straightaway.

Hence, the goal of my platform is to:

- Harness massive volumes of Twitter streaming data.
- Create fast, reliable, and scalable real-time analytics on that data.
- Transfer streaming data to storage and allow this historical data to be analyzed as well.

I embrace the structure and constraints of the Lambda Architecture (explained in Section 2.2.6) when designing EPIC Real-Time. Apache Storm and Hadoop are the technologies proposed by Marz for real-time and batch processing respectively in his book (Marz & Warren, 2015). However, I apply only Apache Spark, an open-source analytic engine for fast large-scale data
processing, to implement both the batch and speed layers for my system. Spark (explained in Section 2.2.7) fits very well with the Lambda Architecture because it provides the same implementation to develop both streaming and batch computations, hence, offering code reuse and reducing the efforts of implementing and maintaining two different systems. So, I developed, tested, debugged, and operated the whole system using a single processing framework.

I also leverage the power of many modern open source technologies in stream processing. For reliable distributed data messaging and queueing, I use Kafka (http://kafka.apache.org), an open-source high-throughput, distributed, publish-subscribe messaging system. For storing the real-time and batch views generated by Spark, I use Cassandra, a large-scale distributed database system.

![Lambda Architecture in EPIC Real-Time](image)

**Figure 11: Lambda Architecture in EPIC Real-Time**

The system architecture is shown in Figure 11. The distribution layer uses Kafka to handle streaming data from Twitter and distributes it to the other layers. The batch layer uses Spark to performs all analytical jobs as batch-oriented computations for a specified time window (e.g. every 5 minutes) creating batch views that are used by the serving layer. The speed layer uses Spark Streaming to process the most recent data stream regularly for a specified time window drastically shorter than the batch layer’s window (e.g. every 5 seconds) producing real-time views for the serving layer. The serving layer retrieves all batch and real-time views from Cassandra, merges
them and serves them to the user through a RESTful API, resulting in different types of user outputs, such as web interfaces of dashboards or command-line interfaces.

My real-time analytics architecture in EPIC Real-Time, discussed above, could also be applied to analyze historical past datasets previously collected for many disaster events in Project EPIC.

### 3.5.2 Providing Access to Analysts

One goal of this thesis is to enable analysts in Project EPIC to perform big data analytics in real-time on streaming Twitter data. EPIC Real-Time aims to provide analysts with basic and advanced functionalities of data analysis with minimal effort. As discussed above, I leverage Spark to implement EPIC Real-Time. However, Spark is targeted at developers only, which means writing code, testing, deploying, and monitoring. Moreover, Spark does not magically clean data, solve analysis problems, or automatically orchestrate workflows. So EPIC Real-Time creates a new service layer on top of Spark to connect the needs of the analysts with the power of Spark.

Therefore, I designed EPIC Real-Time such that analysts can create queries without the need for specifying Spark programs. Instead, EPIC Real-Time provides a layer of abstraction behind a standard set of web interfaces that will be used by analysts to access all platform functionalities. Figure 12 depicts the desired analysts’ interactions with the platform when creating a new query upon streaming data.

![Figure 12: Query Request in EPIC Real-Time](image)

Figure 12: Query Request in EPIC Real-Time
Via a web-based dashboard, an analyst fills in an input form for the type of query of interest. The query request is sent to a Query Constructor module using a RESTful API to interpret the query DSL and generate a new query job. A Query Job is created as a Spark job, then deployed and run in the distributed Spark framework. The Query Job listens to the relevant incoming data streams, processes the job, and stores its result into a data store as a real-time Query View. The query result is then sent to the analyst as a time window-based continuous stream.

Analysts can interact with analytical results in an online web-based dashboard. Online dashboards create unified analytics interfaces for both live events and historical past events. A single dashboard is created for every event. I list below the main features that analysts can apply when working with EPIC-Real-Time

- Create new events.
- Create queries on streaming data.
- Create queries on historical data.
- Modify or delete any running queries.

3.5.3 Engineering a Lightweight Scalable Big Data Platform

This research tackles some important issues related to developing highly scalable, distributed software applications for big data analytics. I created microservices to implement EPIC Real-Time in a modular fashion to produce a system that is easy to deploy and maintain. By adopting microservices, I was able to create independent, loosely-coupled, reusable components in EPIC Real-Time. The queries discussed in the previous section are deployed as microservices and could be run on any node of EPIC Real-Time’s computational infrastructure to connect to Spark, submit their jobs, and process the results.

Furthermore, to support my prototype, I investigated efficient techniques to develop event-driven microservices architectures that control all system workflows, and manage processing and routing of asynchronous events. Event-processing techniques provide asynchronous messaging, transformation, content-based routing, and publish/subscribe mechanisms (Krause, 2015). All these capabilities enabled me to manage multiple flows of streaming data. I utilized Akka's Actor Model (explained in Section 2.3.4) for implementing these features in EPIC Real-Time.
Figure 13: EPIC Real-Time Components

The main components of EPIC Real-Time are shown in Figure 13. At the lowest layer, I have Cassandra and Spark for data persistence and computation at scale. I use Apache Spark to implement both the batch and speed layers in the system. For storing the real-time and batch views generated by Spark, I use Cassandra. For distributed data messaging and queueing, I use Kafka to reliably handle streaming data from Twitter. Then on top of Spark and Cassandra, I implemented actor-based microservices using Akka. I used Akka actors to build two components of EPIC Real-Time: The Events system and the Query engine, as shown in Figure 13. I explain the actor-based design and implementation of these components in Chapter 4. At the top of Figure 13, is EPIC Real-Time’s user interface layer consisting of web apps and command-line applications.
In Figure 14, I explain the scenario of handling a query. First, an analyst creates a new event of interest with a list of relevant keywords. Then, when the event is activated, Twitter's streaming API is used to collect all tweets containing the event’s keywords. The streaming data flows into Kafka. All collected data is persisted immediately in Cassandra as historical data.

![Lambda Architecture for Trending Hashtags](image)

**Figure 14: Query Process Flow (e.g. Trending Hashtags)**

To start exploring the event, an analyst can submit a query such as “Find the top 10 hashtags found in the data set collected over the last week”. The batch layer of the Lambda Architecture responds to this query. The batch layer initiates a Spark job to read the event’s historical data over the past week, runs the query on the retrieved data, then provides a batch view to the serving layer. Simultaneously, the speed layer initiates a Spark Streaming job to process the same query over the most recent data that has arrived from the stream using a shorter time window (e.g. every 5 seconds); this job will provide a new real-time view to the serving layer. The serving layer retrieves both batch and real-time views persisted as Cassandra tables, merges them, and serves them to the user through RESTful APIs via the user interface layer.
3.6 Research Contribution

The main contributions of my research are to support analysts in exploring datasets in real time by creating a prototype that fulfills the requirements I received from my analysts. I identify a set of concepts to make real-time and batch queries easy to create and deploy in a big data analytics environment. I also identify the techniques and frameworks needed to create a prototype that implements these concepts with acceptable performance.

The new prototype—EPIC Real-Time—presents the design and implementation of a scalable, flexible, and extensible real-time big data analytics platform. The platform is useful for building big data applications. By providing a core set of generic real-time and batch queries—while being able to easily and efficiently add new types of queries—my platform creates a set of building blocks that can be used to build big data applications for other application domains. It is EPIC Real-Time’s ability to support a wide range of query types and its ability to quickly deploy those queries onto a cluster for both real-time analysis of streaming data as well as the batch processing of historical data that is my main research contribution. Other systems focus on data models to support this type of analysis or they focus on one type of analysis over the other; my system is unique with respect to its support for flexible, extensible queries for both types of analysis.

While my own application domain is crisis informatics, EPIC Real-Time is capable of supporting a broad range of application domains. It already supports a wide range of commonly-needed features for all data analysts. All that is needed to support a new domain is the integration of new data sources and the query types that support them. Due to my abstractions, adding new query types is straightforward; existing queries are often implemented in less than 40 lines of code.

In the next chapter, I present the design and implementation of a set of generic services that meet the analytical requirements of Project EPIC’s analysts and cope with the challenges of leveraging real-time analysis within the context of crisis informatics research specifically and other problem domains generally. EPIC Real-Time provides a set of REST APIs to access these services and create different types of queries for different trending types. Analysts can create queries and manage the overall analytical process using these APIs.
CHAPTER 4

ARCHITECTURE AND IMPLEMENTATION

EPIC Real-Time implements two REST-based microservices. The Events service allows the analyst to create new events and associate keywords with those events. The Query Engine allows the user to generate and run a wide range of queries on an event’s Twitter dataset. This chapter discusses the architecture and design of EPIC Real-Time in detail, describing its services and touching on topics related to data modeling, API design, analysis pipelines, its programming model, and its supported user interfaces while highlighting many of the platform’s features.

4.1 Events Service

The Events service is built to allow analysts to create and manage events in EPIC Real-Time. Indeed, the first step to use EPIC Real-Time is to create an event. In the crisis informatics domain, an event is a disaster event that an analyst would like to follow and explore in real time via streaming Twitter data. The Events service allows users to add keywords to events to include relevant tweets. Analysts create an event, add keywords, then activate the event to start collecting and persisting data for that event. Analysts can also deactivate an event to stop data collection and they can delete an event if it is no longer needed. For each active event, the Events service collects Twitter data streaming into Kafka and simultaneously consumes and persists this data in Cassandra. Analysts can add keywords at any time after the creation of an event. The Events REST API can be used to provide the following services:

- Add/View/Delete Events.
- Add/View/Delete Event Keywords.
- Activate/Deactivate Events.
The Events Service is illustrated in Figure 15. It consists of four Akka actors that collaborate with each other to create all the logic required for creating and managing events (concurrently). In Akka, an ActorSystem is first created to contain all other actors. For the Events service, the ActorSystem then creates a RestApi router that creates a set of routes to handle HTTP requests and provide an HTTP response. The routes define how HTTP requests should be handled using a convenient DSL, provided by the Akka-http module. The RestApi router creates an actor called the EventSupervisor that supervises all other event-related actors. HTTP requests are sent from the RestApi router and translated to messages that the EventSupervisor actor can handle. The EventSupervisor accepts and processes requests for creating and managing events. It creates a child actor called Event for every new event requested by a user. The EventSupervisor is responsible for sending all requests received for events to their corresponding Event actors.

The Event actor maintains event data in a Cassandra table called “events” where all events are stored, updated, and retrieved by an event’s id. An event’s data include an event’s name, an event’s
description, a start date, an end date, a flag that indicates whether an event is currently active, and a list of the event’s keywords. Once an Event actor is created for an event, it creates a child actor called the TweetKafkaCollector for that event. The TweetKafkaCollector actor creates a new Kafka topic for the event, named by the event, and a Kafka producer client to post tweet messages collected from Twitter's Streaming API into the newly created event topic. The event’s Kafka topic receives all incoming tweet messages collected for the event. The Event actor at the same time creates another child actor called the TweetCassandraWriter that creates a Kafka consumer client for the event’s topic. The TweetCassandraWriter actor, once created, starts to listen to the event’s topic, reads all incoming tweets, and writes them into a Cassandra table called “tweets”. The two actors TweetKafkaCollector and TweetCassandraWriter are created when an event is created. However, they do not start their work until a user activates the event.

4.2 Query Engine

The Query Engine is created to generate and manage queries of three types: Dataset, Trend, and Popular. The engine’s REST API is used to manage queries for an event of interest as follows:

- Add/ View/Delete Queries.
- Start/Stop Queries.
- View Query Results.

4.2.1 Query Request

Creating queries is straightforward, made possible via a standardized interface. Each query request contains a set of parameters including: a query type, a lambda type, and then parameters specific to the previous two selections. These parameters may include a start date, an end date, a threshold/top k value, and a limit value. A query's start and end timestamps determine the type of query to run. If an analyst leaves both fields blank, then the query engine creates and executes just a real-time query. If an analyst supplies both dates, then only a batch query is run to retrieve answers from just that timespan. If an analyst supplies just a start date, then both a batch and real-time query is executed. We can think of this as a since query: “Give me an answer to this query from this start date to the present moment.” Figure 16 illustrates the three different ways to request a query using a command-line tool as well as the corresponding Web user interface for determining
the query’s date option. Figure 16.a shows an example of requesting a real-time query expressed via the “Now” option. Figure 16.b shows an example for requesting combined real-time and batch query expressed via the “Since” option. Figure 16.c shows a third example for requesting a batch query for a limited time duration specified via the “Date Period” option.

The design and implementation of the Query Engine is illustrated in Figure 17. The first actor created by the ActorSystem is called a QuerySupervisor. The QuerySupervisor responds to all query requests generated by a user for an event. The first request defines a query type. For each query type request, the QuerySupervisor creates a child actor called QueryActor. QueryActors are responsible for handling further requests received for their corresponding query types. Each QueryActor responds to requests that define different query attributes.
These attributes are hashtag, user, url, and user mention for Trend and Dataset queries and tweet and user for Popular queries. To handle the different roles related to the Lambda Architecture (batch, speed, and serving), a QueryActor creates a child actor called a LambdaSupervisor. Every created LambdaSupervisor then is responsible for receiving query requests of the corresponding Lambda Architecture role and query type.

![Query Engine Diagram](image)

**Figure 17: Query Engine**

When the user requests a new query, the assigned LambdaSupervisor creates a new actor called LambdaActor that eventually represents and serves the requested query. The LambdaActor created for each requested query holds the query parameters entered by the user (startDateDateTime, endDateDateTime, threshold/topk, and limit value). When a query is submitted to a new LambdaActor, its LambdaSupervisor parent generates a new query ID that is sent to the user to be used in any
further messages to the created query. The query ID consists of the event name, the query type, the lambda type, and a counter number, e.g., USE2016_trend_hashtag_01.

The LambdaActor, as its name implies, generates automatically a complete Lambda architectural design process for its assigned query. As Figure 17 shows, it automatically creates three child actors: SpeedActor, BatchActor, and ServingActor, to process the query and generate its result. For every LambdaActor, users can request to start or stop its process.

The SpeedActor works in the speed layer of its Lambda process. It is responsible for creating a real-time view of the query. It listens to the Twitter data streaming in the event’s Kafka topic in real time and creates a new incremental view with every new window of data. On the other hand, the BatchActor works in the batch layer of its Lambda process, in which a batch query is run continuously upon the historical dataset. Based on the time period selected, the BatchActor reads the required tweet data from Cassandra then generates a new batch view every cycle. The time required for batch processing depends on the time period requested and the size of the dataset. The ServingActor represents the serving layer in the Lambda process. It merges results from the real-time and batch views created by the SpeedActor and BatchActor. The ServingLayer then serves the final results to a user when requested.

The Query Engine relies on the Akka actor model to process and manage all analysis tasks. As implied by the previous diagram, each query submitted by the user spins up a set of actors; this means that, at any point in time, a tree of actors exists each managing jobs in Spark. It is those jobs that actually process the incoming and/or historical data to produce analysis results. Figure 18 illustrates a run-time snapshot showing a hierarchical tree of actors and their relationships serving different types of queries. This tree is created concurrently for each active event.

The Query Engine generates REST APIs to provide access to the engine’s functionality. This API layer creates the engine’s end-user interface to be used either by developers or analysts. These RESTful Web services are implemented efficiently using Akka’s http modules. Figure 19 shows the URL structure of the REST API and demonstrates how each path in the URL is served and supervised by different types of actors. Each feature, provided by a specific actor, is available by sending a GET, POST, or DELETE request using an appropriately formatted URL. For example, POSTing a query causes it to be created while GETting a query retrieves its most recent answer as held by an actor working in the serving layer of that query.
Figure 18: Actors at Run Time

Figure 19: REST API URL Structure
4.3 Data Modeling

Apache Cassandra is a primary component for building data-intensive software systems. Cassandra is a scalable NoSQL database for storing substantial amounts of structured, semi-structured, and unstructured data. Cassandra stores data across all nodes in a cluster automatically. Its replication system allows data to be accessed from any node or restored if any node in the cluster goes down. This guarantees fault-tolerance and avoids a single point of failure. Cassandra provides the capability of scaling horizontally by simply adding nodes to an existing cluster (https://cassandra.apache.org).

My prototype relies heavily on Apache Cassandra for its data model. In my system, the same schema is used for streaming and historical data. I make use of the Spark Cassandra connector from DataStax (https://github.com/datastax/spark-cassandra-connector) to get data out of Spark Streaming and into Cassandra. I need to carefully design my data model to ensure my streaming jobs in Spark are efficient. To do that, I must make appropriate use of Cassandra’s concepts of primary, partition, and clustering keys. These keys determine how data gets distributed across a cluster of machines. The primary key is made up of one or more columns. The first column plays the role of the partition key (also known as the row key) and the remaining columns are known as clustering keys. To achieve uniform distribution, the partition key should be as random as possible; it determines where a row of data is stored within the cluster. The clustering keys are then used to determine how data is stored for a row to ensure that data most frequently used together is stored in physical proximity on a storage device. Secondary indexes in Cassandra provide a way to query a column without using a primary key. I can create a new index for any column. Indexing in Cassandra can greatly improve performance. My system’s Cassandra tables and their associated schemas are described next.

Keyspace and Main Tables

I define my keyspace, called epic_realtime, with the “NetworkTopologyStrategy” replication strategy and specify a replication factor of three since my deployment environment made use of three Cassandra nodes. (See Section 4.4.4 for more information about my cluster configuration.) The Events table is a simple table used to store data about all events in the system, accessed by the event_name column as a primary key. The Tweets table is where millions of tweets are stored and retrieved for many different events. I defined a compound primary key of event_name as a partition
key and created_at and tweet_id as first and second clustering keys. As a result, event_name is used to distribute data across nodes such that tweet rows with adjacent created_at time values of the same event are stored in a single node if possible.

<table>
<thead>
<tr>
<th>Keyspace and Main Tables</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE KEYSPACE IF NOT EXISTS epic_realtime WITH REPLICATION = {'class':&quot;NetworkTopologyStrategy&quot;, 'Cassandra':3};</td>
</tr>
<tr>
<td>CREATE TABLE IF NOT EXISTS epic_realtime.events (</td>
</tr>
<tr>
<td>event_name text PRIMARY KEY,</td>
</tr>
<tr>
<td>event_description text,</td>
</tr>
<tr>
<td>start_date text,</td>
</tr>
<tr>
<td>end_date text,</td>
</tr>
<tr>
<td>is_active boolean,</td>
</tr>
<tr>
<td>keywords text);</td>
</tr>
<tr>
<td>CREATE TABLE IF NOT EXISTS epic_realtime.tweets (</td>
</tr>
<tr>
<td>event_name text,</td>
</tr>
<tr>
<td>created_at text,</td>
</tr>
<tr>
<td>tweet_id text,</td>
</tr>
<tr>
<td>tweet_json text,</td>
</tr>
<tr>
<td>PRIMARY KEY (event_name, created_at, tweet_id));</td>
</tr>
<tr>
<td>CREATE INDEX IF NOT EXISTS tweets_tweet_id ON epic_realtime.tweets (tweet_id);</td>
</tr>
</tbody>
</table>

**Table 3: Cassandra Table Schema - Keyspace and Main Tables**

I defined the event_name column as a partition key instead of the unique tweet_id, because I heavily filter tweet data based on events. I also filter tweets based on their creation times and this primary key allows me to efficiently retrieve all tweets collected for an event that were posted during a specific period of time. As a result of using tweet_id as a second clustering key, tweets are stored in rows in lexicographic order on the same node. In order to be able to retrieve a tweet by its unique id, I had to create a secondary index for the tweet_id column. The schema of my keyspace, Event and Tweet Cassandra table is described in Table 3.

**Query Meta Data Tables**

Working with the Query Engine generates a lot of data that needs to be available for data reuse. For user convenience, I persist all data a user generates using the REST API. I persist data about the query types and lambda types a user requests to save time when generating the same query
again. I maintain a table called “query_types” for the requested query types of an event, and another table “lambda_types” for the requested lambda types. A third table, called “queries”, is used to store all queries requested by users. These metadata tables allow EPIC Real-Time to regenerate a required hierarchy of actors, such that users get the same configuration for a query independent of whether they are using my REST API via a command-line tool or via my Web-based user interface (discussed in Section 4.6). The schemas of my query metadata tables are described in Table 4.

<table>
<thead>
<tr>
<th>Query Meta Data Tables</th>
</tr>
</thead>
</table>
| CREATE TABLE IF NOT EXISTS epic_realtime.query_types(
  event_name text,
  querytype text,
  PRIMARY KEY (event_name, querytype)); |
| CREATE TABLE IF NOT EXISTS epic_realtime.lambda_types(
  event_name text,
  querytype text,
  lambdatype text,
  PRIMARY KEY ((event, querytype), lambdatype)); |
| CREATE TABLE IF NOT EXISTS epic_realtime.queries(
  event_name text,
  querytype text,
  lambdatype text,
  query_name text,
  lowertime text,
  uppertime text,
  counttype text,
  value text,
  running boolean,
  speed_job_id text,
  batch_job_id text,
  serving_job_id text,
  PRIMARY KEY (event, querytype, lambdatype, query_name)); |

Table 4: Cassandra Table Schema - Query Meta Data Tables
Query View Tables

For every query generated by users, the Query Engine creates and maintains three Cassandra tables, named by the query name, for three generated views: real-time, batch, and merged (as discussed in Section 4.2). The continually updated results in the real-time and batch view tables are always available for the serving layer to get, merge, and then serve to a user. The schema of those tables is very simple consisting of two columns, the lambda-type attribute values and their generated count values. Both columns are defined as a compound primary key, representing a unique attribute value and its count. The count column is used as a clustering key that stores data in descending order which is the default ordering for a query result. The schema of my query view tables is described in Table 5 (shown for a Trend hashtag query).

<table>
<thead>
<tr>
<th>Query View Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE TABLE IF NOT EXISTS epic_realtime.${query_name}_realtime (hashtag text, count int, PRIMARY KEY (hashtag, count)) WITH CLUSTERING ORDER BY (count DESC);</td>
</tr>
<tr>
<td>CREATE TABLE IF NOT EXISTS epic_realtime.${query_name}_batch (hashtag text, count int, PRIMARY KEY (hashtag, count)) WITH CLUSTERING ORDER BY (count DESC);</td>
</tr>
<tr>
<td>CREATE TABLE IF NOT EXISTS epic_realtime.${query_name} (hashtag text, count int, PRIMARY KEY (hashtag, count)) WITH CLUSTERING ORDER BY (count DESC);</td>
</tr>
</tbody>
</table>

Table 5: Cassandra Table Schema - Query View Tables
4.4 Analysis Pipelines

The Query Engine provides answers to a wide range of queries over streaming and historical datasets. In this section, I discuss my analysis pipeline in more detail.

4.4.1 Kafka Messaging System

I use Kafka (https://kafka.apache.org) as a fast, scalable, reliable, publish-subscribe messaging system. In Kafka, a stream of messages of a particular type is called a topic. The topics in EPIC Real-Time contain streams of tweet JSON objects collected for events. Kafka is horizontally scalable and can partition its topics across multiple servers for fault tolerance. Spark connects to Kafka using a built-in API to create tweet data streams for real-time queries. Kafka offers high throughput for both publishing and subscribing, such that multiple consumers retrieve their own copy of a message. Moreover, consumers can consume messages at their own speed. If an exception occurs while consuming a message, the consumer can ask for the message again. These features allow EPIC Real-Time to create multiple consumers for a topic, one for each activated real-time query. I also create a consumer that ensures that all tweets for each event get persisted to Cassandra to serve as the historical datasets for batch queries.

4.4.2 Spark Processing Jobs

As explained above, the LambdaActors are responsible for generating a process based on the Lambda Architecture. The actors in this process connect to my Spark cluster, delegating to the Spark Core and Spark Streaming engines to process queries. These actors connect programmatically to Spark using its REST API. This API allows us to submit Spark jobs, check their status, or cancel them. Once a job gets accepted by Spark, the API returns a job’s driver id, which I can use to kill a job if a user cancels a query.

The speed layer for a query (Figure 20.a) runs a Spark Streaming job for processing and rendering query result in real time. It reads all tweet objects arrived in the event’s Kafka topic as streaming Twitter data, looking for values for the requested attribute. The streaming job reads tweet objects as Spark Dstreams and then extracts and aggregates the discovered values computing their accumulated number of occurrences. The streaming job then stores its result as a real-time view of value and count tuples in a newly generated Cassandra table. Table 6 shows a code snippet
of the speed layer implemented by Spark. The speed layer keeps updating counts or appending rows of new values to the real-time view every 5 seconds. The process of updating values can be expensive but luckily it is done on a small set of data. The speed layer may experience some data loss when a data stream experiences high volume, however, its results are corrected when it is replaced with a batch layer view, discussed next.

![Spark Streaming Diagram]

**Figure 20: Spark Lambda Process**

Asynchronous to the speed layer, the batch layer (also in Figure 20.a) runs the same job as the speed layer but on stored data. The batch job runs a Spark job for processing batch queries upon the tweet data stored in Cassandra. It reads tweet objects as Spark RDDs for a specified small or large date range of tweet creation times. It extracts and aggregates values of the requested attribute with their accumulated number of occurrences. It then stores its result as a batch view of value and count tuples in a newly generated Cassandra table. The batch job repeats its process continually in a while loop, unless it is halted by the user. This continuous loop ensures that the batch view
contains a complete and correct view of the entire dataset. Table 7 shows a code snippet of the batch layer implemented by Spark. Moreover, at the end of every loop iteration, the batch job appends all rows found in the corresponding real-time view table to its batch view table, then deletes all rows from the real-time view. The schema of the Cassandra tables guarantees that no duplication of rows happens when appending real-time data to batch data.

Asynchronously again, the serving layer (Figure 20.b) works upon the data produced by the speed and the batch layers. The serving layer runs a Spark Streaming job that creates Dstreams (every 5 seconds) over data read from both the real-time and batch views. The serving job then merges and aggregates all data found in the two views, and make the final results available to the user in real time. Table 8 shows a code snippet of the serving layer implemented by Spark. Figure 20 also shows a fourth layer, not part of Spark, called the query layer (Figure 20.c). This layer stores a continuous query that is served through the REST API or the Web user interface. The query layer provides the final generated query result via a stream that is rendered to the user in real time. Real time updates of query results vary in response time. The response time of a streaming query (3 seconds average) is affected by the size of the data streaming into the system per second, while the response time of a batch job (from few seconds to few minutes) is mostly based on the size of the historical data requested in the query.
//Connect to Kafka using Spark’s Kafka APIs
val directKafkaStream = KafkaUtils.createDirectStream[String, String, StringDecoder, StringDecoder](ssc, kafkaConf, topicsSet).map(_._2)
directKafkaStream.foreachRDD(rdd => {
  if (rdd.count>0){
    // Read received RDDs as Spark’s DataFrame (for easier reading of JSON objects)
    val df = sqlContext.read.json(rdd)

    // Extract tweets’ user values
    val users = df.select(df("user.screen_name"))

    // Aggregate user values with their number of occurrences
    // Save into Cassandra as query’s real-time view
    users
      .map(x => (x.getString(0),1))
      .reduceByKey(_+_
      ).saveToCassandra("epic_realtime",
      s"${query_name}_realtime",
      SomeColumns("user", "count"))
  } //if-end
} //directKafkaStream-end

<table>
<thead>
<tr>
<th>Spark Job: Speed Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Code Example:</strong> Query type = Trend, Lambda type = User</td>
</tr>
</tbody>
</table>

Table 6: Spark Job Example of Speed Layer
Spark Job: Batch Layer

Code Example: Query type = Trend, Lambda type = User

// A continues while loop reading tweet data from Cassandra
while (true){
    // Get tweets filtered by event’s name and a date range
    val tweetRDD = sc.cassandraTable("epic_realtime", "tweets")
        .select("tweet_json")
        .where(date_filter)
    // Extract tweets’ user values
    val users = tweetRDD.map{ row =>
        Json.parse(row.getString("tweet_json")
            "user")
    }
    // Aggregate user values with their number of occurrences
    // Save into Cassandra as query’s batch view
    users
        .map(x => (x, 1)).reduceByKey(_+_)
        .saveToCassandra("epic_realtime",
            s"${query_name}_batch",
            SomeColumns("user", "count"))
} //loop-end

Table 7: Spark Job Example of Batch Layer
Spark Job: Serving Layer

Code Example: Query type = Trend, Lambda type = User

```scala
// Read and combine data from real-time and batch views
val realtimeRDD = sc.cassandraTable("epic_realtime", s"${query_name.toLowerCase()}_realtime")
val batchRDD = sc.cassandraTable("epic_realtime", s"${query_name.toLowerCase()}_batch")
val unionRDD = realtimeRDD.union(batchRDD)

// Generate continuous stream of Dstream
val dstream = new ConstantInputDStream(ssc, unionRDD)
dstream.foreachRDD{ rdd =>
  if (realtimeRDD.count>0 || batchRDD.count>0){
    // Aggregate user values with their count values
    val userCount = rdd.map(row =>
      (row.getString("user"), row.getInt("count")))
      .reduceByKey(_+_)
    // Convert to Spark’s DataFrame (for data
    .toDF.select(col("_1").alias("user"),
      col("_2").alias("count"))
    val trendUser = trend match {
      case "threshold" => userCount.where("count">=value)
      case "topk" => userCount.limit(value.toInt)
      case _ => userCount.limit(10)
    }

    // Save results as a query result view
    trendUser.write.format("org.apache.spark.sql.cassandra")
      .options(Map("table" -> s"${query_name}",
        "keyspace" -> "epic_realtime")
    .mode(SaveMode.Overwrite)
      .save()
  }
//dstream-end
```

Table 8: Spark Job Example of Serving Layer
Moreover, to demonstrate how the Lambda Architecture works to update query results, Figure 21 shows timeline examples of query updates when combining results of batch and real-time views. Since a real-time view is intended to be transient, it gets discarded as soon as a copy of its data is appended to a batch view. As demonstrated in Figure 21, when one queries a newly created event, one needs, e.g., 5 seconds to get a real-time update and 100 seconds to receive an update from the batch view. The batch processing time will get longer as the dataset gets larger throughout the lifetime of an event. However, the growth of an event’s dataset does not affect the performance of real time updates, since those are always operating on the most recently arrived data.

Figure 21: Lambda Update

4.4.3 Spark Cassandra Connector

All three Lambda Architecture layers—speed, batch, and serving—are empowered by fast in-memory distributed Spark jobs. All these jobs connect to Cassandra either for data write (in the case of the speed layer) or data read and write tasks (in the case of the batch and serving layers), in which all generated data views are persisted as fast-accessed distributed Cassandra tables. To integrate between these two different paradigms of data handling, I leverage the Spark Cassandra Connector, an open-source library developed by DataStax (https://github.com/datastax/spark-cassandra-connector). The connector provides an easy and efficient ways to transfer data between
Spark and Cassandra. It reads Cassandra tables as Spark RDDs and write Spark RDDs to Cassandra tables, while providing APIs to run arbitrary CQL queries in Spark applications. Furthermore, in most cases, the connector saves EPIC Real-Time from running expensive Spark data shuffling and grouping, in which Spark data partitioning is applied throughout the Spark cluster. With the connector, whenever it is possible, I leverage Cassandra to filter and sort data in its cluster where data is already partitioned and processed locally in different distributed nodes. This takes advantage of data locality (Spitzer & Chan, 2015). In fact, when using DataStax Enterprise, I guarantee the use of data locality since Cassandra’s data partitioning operations and relocation of Spark RDDs is aligned with the replication strategy in my Cassandra Keyspace. This allows Spark and Cassandra to work together to create efficient big data processing.

4.4.4 Cluster Configuration

I use DataStax (http://www.datastax.com) v.5.0.1 to create and manage my Cassandra and Spark nodes. My cluster consists of three Cassandra nodes and three Spark nodes created in a private open-stack cluster, each with 16 VCPUs, 32GB RAM, and 250GB attached volumes. A Kafka cluster of three brokers is used to provide distributed reliable data streaming from Twitter APIs to my system.

4.5 Programming Models

I built EPIC Real-Time with REST-based services implemented by Akka’s http and actor models. Akka helped me to create concurrent, distributed services with asynchronous and non-blocking message communications. Similar to object oriented programming, actors are interoperable self-contained objects each responsible for specific assigned tasks. With parent-child relationships, Akka actors allowed me to decompose large tasks into small ones assigned for dedicated actors that concurrently process tasks and wait for new ones. Actors can create other actors, send messages to each other, and respond to incoming messages accumulated in their mailboxes. Actor-based programming works well with applications that need to scale rapidly (Roestenburg, et al., 2016).
4.6 RESTful Services and API Design

The EPIC Real-Time platform is meant to provide data exploration and analytical tools for large volumes of data sets that would otherwise be difficult for analysts to do in easy and accessible ways. My platform provides RESTful Web services for analytical queries on data streaming into the system. Those services provide a set of tools to analysts’ hands while hiding the technical complexity of data integration and processing. With REST-based services, I make use of a standard software design for industry and enterprise applications. With my platform’s end user web services, I can provide standard and accessible ways for an analyst to explore big data in real time and to run batch jobs as well. For my developers, these services can be used programmatically to build applications on top of them. With respect to end users, they are simple to use and understand without the need of installing heavyweight frameworks. With respect to the software architects, they are easy to implement, maintain, and extend. RESTful services allow for APIs to be created for programmatic use in web based applications and for direct use by command line tools.

The design of RESTful services is based on resources and how to provide access to these resources, enabling Resource-Oriented Architecture based services (Richardson, et al., 2007). In my system, I identified different types of resources and determined the relationship between these resources. The design of a service API is very important and critical. I designed my service APIs such that they are easy to learn, easy to use, and easy to extend. I designed my APIs such that they are used to access data resources that represent important assets for analysts. The RESTful principles, first introduced by Roy Fielding (Fielding, et al., 2000), state that one identifies logical resources as nouns, not verbs, for internal models such as events, keywords, and tweets. Then one defines actions on these resources. The RESTful principles provide CRUD actions on resources leveraging existing HTTP methods (GET, POST, PUT, DELETE). REST APIs create standard HTTP responses. Each response contains a status code that represents the status of the request, such as 200 OK or 201 Created. The main body part of the response contains the output data that is sent to a user.

When designing my REST APIs, I followed the main design principles (Richardson et al., 2007). “Events” are the first resource my system provides to access and manage an event’s data. The basic types of resources are collection resources, such as /events, and instance resource, such as /events/USE2016, that relies on an identifier (event_id) known by the end users. The other
resource type is the queries EPIC Real-Time provides to end users from the Query Engine. The API URL structure follows the same hierarchy as my query design (shown in Figure 19), making queries easy to generate and access using unique URLs. I use JSON as my main format to send requests and receive responses.

In Tables 9 and 10, I provide description of all my service APIs. This type of documentation is accessible to my analysts and developers.

**Table 9: Events Service API Description**

<table>
<thead>
<tr>
<th>Description</th>
<th>HTTP Method</th>
<th>URL</th>
<th>Request Body</th>
<th>Status Code</th>
<th>Response Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create an event</td>
<td>POST</td>
<td>/events/USE2016</td>
<td>`{</td>
<td>201 Created</td>
<td><code>{&quot;name&quot;: &quot;USE2016&quot;, &quot;description&quot;: &quot;US Precedential Election 2016&quot;, &quot;startDate&quot;: &quot;2016-11-01&quot; }</code></td>
</tr>
<tr>
<td>Get an event</td>
<td>GET</td>
<td>/events/USE2016</td>
<td>N/A</td>
<td>200 OK</td>
<td><code>{&quot;name&quot;: &quot;USE2016&quot;, &quot;description&quot;: &quot;US Precedential Election 2016&quot;, &quot;startDate&quot;: &quot;2016-11-01&quot;, &quot;endDate&quot;: &quot;, &quot;isActive&quot;: false }</code></td>
</tr>
<tr>
<td>Get all events</td>
<td>GET</td>
<td>/events</td>
<td>N/A</td>
<td>200 OK</td>
<td><code>{&quot;events&quot;: [ { &quot;name&quot;: &quot;USE2016&quot;, &quot;description&quot;: &quot;US Precedential Election 2016&quot;, &quot;startDate&quot;: &quot;2016-11-01&quot;, &quot;endDate&quot;: &quot;, &quot;isActive&quot;: false }, { &quot;name&quot;: &quot;standingrock&quot;, &quot;description&quot;: &quot;Standing Rock Protests&quot;, &quot;startDate&quot;: &quot;2016-11-08&quot;, &quot;endDate&quot;: &quot;, &quot;isActive&quot;: false } ] }</code></td>
</tr>
<tr>
<td>Cancel an event</td>
<td>DELETE</td>
<td>/events/USE2016</td>
<td>N/A</td>
<td>200 OK</td>
<td><code>{ }</code></td>
</tr>
</tbody>
</table>
### Table 10: Query Engine API Description

<table>
<thead>
<tr>
<th>Description</th>
<th>HTTP Method</th>
<th>URL</th>
<th>Request Body</th>
<th>Status Code</th>
<th>Response Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create an event’s keyword</td>
<td>POST</td>
<td>/events/USE2016/keywords</td>
<td>{ word = trump }</td>
<td>201 Created</td>
<td>{ &quot;text&quot;: &quot;trump&quot; }</td>
</tr>
<tr>
<td>Get an event’s keywords</td>
<td>GET</td>
<td>/events/USE2016/keywords</td>
<td>N/A</td>
<td>200 OK</td>
<td>{ &quot;keywords&quot;: [ { &quot;text&quot;: &quot;trump&quot; }, { &quot;text&quot;: &quot;hillary&quot; } ] }</td>
</tr>
<tr>
<td>Delete an event’s keyword</td>
<td>DELETE</td>
<td>/events/USE2016/keywords</td>
<td>{ word = trump }</td>
<td>200 OK</td>
<td></td>
</tr>
<tr>
<td>Change an event’s active status</td>
<td>POST</td>
<td>/events/USE2016/activate</td>
<td>{ status = true }</td>
<td>200 OK</td>
<td>{ &quot;name&quot;: &quot;USE2016&quot;, &quot;description&quot;: &quot;US Precedential Election 2016&quot; &quot;startDate&quot;: &quot;2016-11-01&quot;, &quot;endDate&quot;: &quot;,&quot;, &quot;isActive&quot;: true }</td>
</tr>
<tr>
<td>Get an event’s tweet count</td>
<td>GET</td>
<td>/events/USE2016/count</td>
<td>N/A</td>
<td>200 OK</td>
<td>{ &quot;count&quot;: &quot;655786&quot; }</td>
</tr>
<tr>
<td>Create a popular lambda type</td>
<td>POST</td>
<td>/events/USE2016/popular/user</td>
<td>N/A</td>
<td>201 Created</td>
<td></td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------</td>
<td>-------------------------------</td>
<td>-----</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>{</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;event&quot;: &quot;USE2016&quot;,</td>
<td></td>
<td>&quot;querytype&quot;: &quot;popular&quot;,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;lambdatype&quot;: &quot;user&quot;</td>
<td></td>
<td>}</td>
<td></td>
</tr>
<tr>
<td>Get all popular lambda types</td>
<td>GET</td>
<td>/events/USE2016/popular/lambdatypes</td>
<td></td>
<td>200 OK</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>{</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;lambdatypes&quot;: [</td>
<td></td>
<td>}</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
<td></td>
<td>}</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
<td></td>
<td>}</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
<td></td>
<td>}</td>
<td></td>
</tr>
<tr>
<td>Create a popular user query</td>
<td>POST</td>
<td>/events/USE2016/popular/user/queries</td>
<td></td>
<td>201 Created</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
<td></td>
<td>{</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>startdate = &quot;&quot;,</td>
<td></td>
<td>&quot;name&quot;: &quot;USE2016_popular_user_01&quot;,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>enddate = &quot;&quot;,</td>
<td></td>
<td>&quot;querytype&quot;: &quot;popular&quot;,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>counttype = &quot;topk&quot;,</td>
<td></td>
<td>&quot;lambdatype&quot;: &quot;user&quot;,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>limit = &quot;10&quot;</td>
<td></td>
<td>&quot;uppertime&quot;: &quot;&quot;,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&quot;counttype&quot;: &quot;topk&quot;,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&quot;limit&quot;: &quot;10&quot;,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&quot;running&quot;: &quot;false&quot;,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&quot;speed_job_id&quot;: &quot;&quot;,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&quot;batch_job_id&quot;: &quot;&quot;,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&quot;serving_job_id&quot;: &quot;&quot; }</td>
<td></td>
</tr>
<tr>
<td>Get a popular user query</td>
<td>GET</td>
<td>/events/USE2016/popular/user/queries/USE2016_popular_user_01</td>
<td></td>
<td>200 OK</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
<td></td>
<td>{</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>name: &quot;USE2016_popular_user_01&quot;,</td>
<td></td>
<td>&quot;name&quot;: &quot;USE2016_popular_user_01&quot;,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>querytype: &quot;popular&quot;,</td>
<td></td>
<td>&quot;querytype&quot;: &quot;popular&quot;,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>lambdatype: &quot;user&quot;,</td>
<td></td>
<td>&quot;lambdatype&quot;: &quot;user&quot;,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>lowertime: &quot;&quot;,</td>
<td></td>
<td>&quot;uppertime&quot;: &quot;&quot;,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>counttype: &quot;topk&quot;,</td>
<td></td>
<td>&quot;counttype&quot;: &quot;topk&quot;,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>limit: &quot;10&quot;,</td>
<td></td>
<td>&quot;limit&quot;: &quot;10&quot;,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>running: &quot;false&quot;,</td>
<td></td>
<td>&quot;running&quot;: &quot;false&quot;,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>speed_job_id: &quot;&quot;,</td>
<td></td>
<td>&quot;speed_job_id&quot;: &quot;&quot;,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>batch_job_id: &quot;&quot;,</td>
<td></td>
<td>&quot;batch_job_id&quot;: &quot;&quot;,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>serving_job_id: &quot;&quot;</td>
<td></td>
<td>&quot;serving_job_id&quot;: &quot;&quot; }</td>
<td></td>
</tr>
<tr>
<td>Get all popular user queries</td>
<td>GET</td>
<td>/events/USE2016/popular/user/queries</td>
<td></td>
<td>200 OK</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
<td></td>
<td>{</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;queries&quot;: [</td>
<td></td>
<td>}</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
<td></td>
<td>}</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
<td></td>
<td>}</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
<td></td>
<td>}</td>
<td></td>
</tr>
<tr>
<td>Start a popular user query</td>
<td>POST</td>
<td>/events/USE2016/popular/user/queries/USE2016_popular_user_02/start</td>
<td>N/A</td>
<td>200 OK</td>
<td></td>
</tr>
<tr>
<td>---------------------------</td>
<td>------</td>
<td>---------------------------------------------------------------</td>
<td>-----</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;name&quot;: &quot;USE2016_popular_user_02&quot;,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;querytype&quot;: &quot;popular&quot;,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;lambdatype&quot;: &quot;user&quot;,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;lowertime&quot;: &quot;2016-11-01&quot;,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;uppertime&quot;: &quot;2016-11-10&quot;,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;counttype&quot;: &quot;threshold&quot;,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;limit&quot;: &quot;100&quot;,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;running&quot;: &quot;true&quot;,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;speed_job_id&quot;: &quot;driver-20161104142349-0000&quot;,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;batch_job_id&quot;: &quot;driver-20161104142349-0001&quot;,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;serving_job_id&quot;: &quot;driver-20161104142349-0002&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Get a popular user query’s result</th>
<th>GET</th>
<th>events/USE2016/popular/user/queries/USE2016_popular_user_02</th>
<th>N/A</th>
<th>200 OK</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>{</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;result&quot;: [</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;user&quot;: &quot;johnson7&quot;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;count&quot;: &quot;145&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>},</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;user&quot;: &quot;us_news&quot;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;count&quot;: &quot;138&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>},</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;user&quot;: &quot;ross_2016&quot;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;count&quot;: &quot;114&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>},</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;user&quot;: &quot;ben_73&quot;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;count&quot;: &quot;109&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>},</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;user&quot;: &quot;adam123&quot;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;count&quot;: &quot;106&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stop a popular user query</th>
<th>POST</th>
<th>/events/USE2016/popular/user/queries/USE2016_popular_user_02/stop</th>
<th>N/A</th>
<th>200 OK</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>{</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;name&quot;: &quot;USE2016_popular_user_02&quot;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;querytype&quot;: &quot;popular&quot;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;lambdatype&quot;: &quot;user&quot;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;lowertime&quot;: &quot;2016-11-01&quot;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;uppertime&quot;: &quot;2016-11-10&quot;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;counttype&quot;: &quot;threshold&quot;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;limit&quot;: &quot;100&quot;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;running&quot;: &quot;false&quot;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;speed_job_id&quot;: &quot;&quot;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;batch_job_id&quot;: &quot;&quot;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;serving_job_id&quot;: &quot;&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cancel a popular user query</th>
<th>DELETE</th>
<th>/events/USE2016/popular/user/queries/USE2016_popular_user_02</th>
<th>N/A</th>
<th>200 OK</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>{</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;name&quot;: &quot;USE2016_popular_user_02&quot;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;querytype&quot;: &quot;popular&quot;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;lambdatype&quot;: &quot;user&quot;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;lowertime&quot;: &quot;2016-11-01&quot;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;uppertime&quot;: &quot;2016-11-10&quot;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;counttype&quot;: &quot;threshold&quot;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;limit&quot;: &quot;100&quot;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;running&quot;: &quot;false&quot;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;speed_job_id&quot;: &quot;&quot;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;batch_job_id&quot;: &quot;&quot;,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;serving_job_id&quot;: &quot;&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4.7 User Interface

When designing EPIC Real-Time, my goal was to build a set of web services that create a scalable and extensible software application that monolithic applications cannot mimic. I developed my real-time and batch-oriented queries as REST services that can be accessible by end users as Web interfaces or directly by developers or analysts using command line tools. The APIs allow reuse of these services through many ways. Developers can build different Web-based applications based on these services. Those applications can be tailored to fit a specific need for their end users.

I also created a simple Web UI that provides high-level access to my services. I used this interface to run continuous queries on the Web, and to allow my analysts to evaluate my new query services. The Web application platform is built using the Play Framework—an MVC framework used for building lightweight but highly-scalable applications (https://www.playframework.com). I also used Akka Streams, a high-level API for asynchronous and non-blocking data streaming built on top of Akka’s existing actor model (http://doc.akka.io/docs/akka-stream-and-http-experimental/current/scala.html) to send query results to the Web application as a series of messages to the user showing a query’s streaming results.

The design of this Web application is simple, allowing analysts to add a new event, view all events available in the system, and navigate to an event’s home page. An event’s home page allows analysts to work on an event of interest. Through the event’s home page, analysts can add/delete keywords and check the number of tweets that have been collected at any moment. There is a side menu showing the available query types; Dataset, Trend, and Popular. For each selected query type, a corresponding set of attributes is shown as horizontal tabs. By selecting a query type and an attribute, analysts can add a new query using a simple Web form. Figure 22 shows a query entry form that looks the same for all query types. Moreover, analysts can find all previously requested queries listed for any specific query type. By simply clicking on a query name, the result is displayed as streaming data updated every second. At any time, analysts can stop running a query or even delete it from the system.
Figures 23 and 24 show two snapshots of the Web UI running two queries in real-time. In Figure 23, I show the results of a Trend query on hashtags; the query was configured to show the top ten hashtags associated with one of my evaluation datasets, the 2016 US Presidential Election. This was a real-time query; the Web UI updates the results every half a second with potentially new hashtags and frequency counts. In Figure 24, I show the results of a Popular User query for this same event, showing the top ten most retweeted users. This also was a real-time query with the results updating every half a second with new users and counts.
Figure 23: Web UI for Trend Hashtag Query

Figure 24: Web UI for Popular User Query
4.8 Platform Features

EPIC Real-Time has many useful characteristics targeted in the design of my platform. Scalability is a critical feature along with low latency, reliability, fault tolerance, and extensibility, especially when building applications in the context of big data (Marz, et al., 2015).

EPIC Real-Time’s scalability is provided by the use of horizontally scalable technologies such as Kafka, Spark, and Cassandra. Spark provides large-scale distributed data processing; adding more computing resources via Spark workers can significantly reduce the processing time of running queries. It also increases parallelism by providing the ability to run queries in parallel. Cassandra can store massive amounts of data reliably via its automatic replication. Moreover, Kafka allows EPIC Real-Time to avoid losing data after it has been collected from Twitter but before it is stored into Cassandra via its reliable message queues.

Fault-tolerance against human errors or system errors is provided by embracing the Lambda Architecture. The concept of data immutability and the ability to run queries over the entire dataset provides an important feature of being able to tolerate errors in the speed layer. That is, if a system error caused a problem in the speed layer, it does not have a huge impact on the system. Since the real-time view produced by a speed actor is transient, any error caused by this layer can be fixed when this data is processed by the batch layer. The same is true with the batch layer. Since the underlying dataset is immutable, a batch view created by a batch actor last only as long as the next batch processing cycle. If a batch actor gets deployed with a bug, only the view that it generates contains bad information. If we fix the bug, the bad batch view gets replaced when the new actor is replaced and used during the next batch cycle.

Extensibility is also provided by leveraging the Lambda Architecture since it allows adding new query types in the future. New queries can be added at any time to both the speed and batch layers. Furthermore, EPIC Real-Time is designed to allow new types of queries to be easily added to the system via the addition of new actors and their associated Spark jobs.

Embracing the Lambda Architecture design also adds additional desirable characteristics to EPIC Real-Time, such as low-latency. The speed layer compensates for the delay of the batch layer providing users with access to the most recently collected data as fast as possible; the batch layer however, as mentioned before, ensures that any mistakes made by the speed layer due to spikes in the data stream are eventually corrected once the next batch processing cycle is complete.
Spark provides fast, in-memory data processing partitioned across a cluster. As a result, EPIC Real-Time can provide even better performance by being deployed on larger clusters of machines. In the next chapter, I present the evaluation methods I applied to evaluate EPIC Real-Time.
CHAPTER 5

EVALUATION

The analysis, design, and architecture of the new platform EPIC Real-Time presented in this dissertation is aimed to result in an accessible and flexible service-based platform for real-time big data analytics. In this section, I describe the work I did to evaluate it. The evaluation is based on two dimensions. On one hand, the system is evaluated on how well the real-time and batch-oriented queries meet the needs of Project EPIC analysts. On the other hand, the system is evaluated on how efficiently it performs its queries. Section 5.1 discusses the feedback collected from my analysts and Section 5.2 describes the performance evaluation.

5.1 User Feedback

The first part of my evaluation consisted of studying how closely EPIC Real-Time meets the needs of the analysts I previously interviewed. This is appropriate since EPIC Real-Time is based on the requirements I elicited from these analysts. This type of evaluation needs to show if the analysts would indeed change the types of analytical questions they ask, and how their research practices could change as a result of the real-time analytical capabilities. To be successful in this dimension, EPIC Real-Time’s new functionalities should support the analysts in performing real-time analysis on large data sets collected from Twitter as a disaster event unfolds. The platform also needs to provide accessible interfaces to these analysts. Hence, I had a subset of my subjects try the features of EPIC Real-Time during an active data collection. In this section, I report on the analysts’ reactions to the prototype and whether it met their expectations. I interviewed three of the seven analysts from my initial investigative study. In the first part of the interviews, I conducted think-aloud sessions to evaluate EPIC Real-Time’s ease of use, real-time response, and flexibility to generate different types of queries. In the second part of the interviews, I collected feedback from the analysts on the platform and how satisfied they were with the new platform. I discuss the two parts of the interviews below.
5.1.1 Think Aloud Protocol Runs

To run my think-aloud sessions, the analysts were asked to use EPIC Real-Time’s Web UI and perform some specific tasks:

- Create a new event or go to an existing one.
- For an event:
  - Add some keywords
  - Activate the event
  - Create a query of any type
  - Run the query

The results of the analysts working on the Web UI were revealing. A prime issue was related to the fact that the design of my Web UI is different than what they are used to in EPIC Collect and EPIC Analyze. In EPIC Collect, they are used to creating a new event and adding new keywords in the same request form. In EPIC Real-Time, the page for creating a new event does not include adding keywords. Instead, adding keywords is an operation that is available from the home page of an event. Also, they found specifying the start date of a new event confusing. They did not know what to enter. Moreover, in EPIC Analyze, analysts are used to exploring and filtering tweets based on various parameters such as date range or language. Unfortunately, I did not have time to implement a filtering feature for EPIC Real-Time. Activating an event was straightforward and they liked the notification that shows the event is active on the header bar of an event home page. This bar also shows the number of tweets that have been collected. Analysts were able to create a new query with minimal guidance. It was clear to them what was meant by the date range options of Now, Since, and Time Period, along with the top k and threshold options.

5.1.2 General Feedback

Analysts were impressed with the streaming results of real-time queries. They valued capturing these results while the event was active especially when compared to EPIC Collect and EPIC Analyze. They found that being able to run these queries in real time was useful since it helped to highlight important features on the disasters they were studying. They were also satisfied with the capabilities of the three query types. They liked that they could easily identify the most active users or the most popular tweets. The analysts wanted to learn more about the REST APIs and how they
could access them directly. The analysts also provided some insights on how these queries could be more useful. They requested the ability to track changes to the query results over time. They suggested for example to get a long timeline of running queries that shows differences of results over the lifetime of an event. The analysts wanted this feature as it would help them consider new paths for their research when analyzing crisis events across different stages of disasters. They would also like to see any type of visualization of result changing in real time, such as moving charts or changing colors of changed data. They also asked for some features that are not implemented yet such as the downloading of query results or samples of the underlying tweet data.

### 5.1.3 Discussion

When I created the Web UI for my analysts, I followed the semantics of the REST service API. The Web UI relied completely on calling my services for all actions. When designing my services, I provided an API call for every function—from creating a query to running a query to deleting a query—with the goal of providing end users with complete control of the available functionality. However, when designing Web interfaces for analysts, it would be easier to just hide these low-level design details. For example, having a button for running a query was confusing. The analysts just expected to see a running query once it was created. Providing end users with a capability to manage queries was meant to enable them to control the process of queries, such that they can stop a running query when there is no need for that query or to allow them to save computing resources. As a result of these follow-up interviews, I found that only some of the services provided by my platform need to be exposed to analysts in the Web interface. I just need to expose what makes sense for them and hide those aspects that create confusion.

### 5.2 Performance Testing

I also evaluated EPIC Real-Time’s performance when processing real-time and batch-oriented queries. High performance, scalability, and reliability are the most important features to evaluate when building data-intensive systems. The Query Engine is the main processor that creates and manages all running queries. Hence, I evaluated the Query Engine in three different scenarios. These scenarios include real-time query performance and batch query performance.
5.2.1 Real-Time Performance

To evaluate the performance of real-time queries, I calculate the required processing time for generating real-time views. The *processing time* is the time that Spark needs to process an RDD window of received tweets. I also track any possible delay that tweets encounter when streaming into the prototype until processing them in real time. I calculate a value called immediacy time to track how long it takes for a tweet to be processed after it was created. It represents the time from when a single tweet is created (using the tweet’s creation time) until the time the tweet is processed by one of the Spark streaming jobs. Figure 25 illustrates how these measurements were calculated. The processing time is a small part of the immediacy time.

![Components of Immediacy Time](image)

**Figure 25: Components of Immediacy Time**

On average, the system takes just 0.7 seconds to process tweets even though it takes 6.7 seconds on average to deliver analysis results to my analysts (see Table 11). The difference between these two values shows that tweets get delayed before getting processed by Spark streaming jobs. The first delay is imposed by Twitter. When a user creates a tweet, Twitter takes on average one second to process it before delivering it to EPIC Real-Time. I calculate this delay by taking the time I receive a tweet in Kafka with the time it was created. The second delay occurs within EPIC Real-
Time. Tweets can spend up to five seconds on average in Kafka before they are finally consumed by Spark. In fact, Spark and Kafka work together to divide tweets up into analysis windows that are turned into Spark RDDs of the received tweets. This delay, however, corresponds to the size of the analysis window specified by EPIC Real-Time. I ask Spark Streaming to send me a batch of tweets every five seconds and so EPIC Real-Time currently imposes only a 0.7 second delay on top of that before delivering results to its analysts. I believe this delay could be addressed by deploying my system on a larger cluster; that would allow me to separate the Kafka nodes from the Spark nodes and would allow more nodes to be allocated both to Kafka and Spark. I would then be able to specify a smaller analysis window and thereby decrease the immediacy time. The following values are the measurements I used to calculate my performance results:

Processing Time =

The time when tweet RDDs are processed - The time when tweet RDDs are received

Immediacy Time =

The time when tweet RDDs are processed - The time when tweet objects are created.

I also evaluated the system on how its performance might change when dealing with high volumes of tweets streaming into the system. I picked two different current events for this evaluation. The first event is relatively recent, the “Standing Rock Protests” event, filtered by three keywords (standingrock, nodapl, and waterislife). It is a relatively slow event in terms of the number of incoming tweets per second. The second event, which experienced high volume, is the “US Presidential Election 2016”, filtered by three keywords (trump, hillary, and president). I measure the performance of Epic Real-Time’s queries across three different query types.

Table 11 provides the performance measurement results when running three types of real-time queries on the two events. All calculated values of processing time—immediacy time, delay, and RDD size—are presented as max, min, and average values. The average processing time is less than a second for all query types in both events. This shows stability of the real-time query although the RDD size ranges from 8 tweets to 240 tweets per stream. Moreover, the average values of immediacy time (~ 6 seconds) are also close for all query types in both events.
Table 11: Performance Measurement of Real-Time Queries

Figure 26 illustrates the processing times along with different RDD sizes for the “US Presidential Election 2016” event. The graphs show that the values of processing time and immediacy time are consistent in the three query types; the processing time is ~ 0.6 seconds and the immediacy time is ~ 6.6 seconds. The graphs show some fluctuation in both processing time and immediacy time, however, the two values are relatively stable most of the time, meaning high values of processing time shows high values of immediacy times, and vice versa.
Figure 26: Real-Time Queries on US Presidential Election


5.2.2 Batch Performance

To evaluate the performance of batch queries, I calculated the processing time required for generating complete batch views while an event is active, such that:

\[
\text{Processing Time} = \text{The time when the processing of tweet RDDs completed} - \text{The time when the processing of tweet RDDs started.}
\]

I evaluated the batch query on the “US Presidential Election 2016” event (id = USE2016), as it grew from ~500K tweets to ~1M tweets. I ran batch queries over a few hours on Election Day, which showed high volume throughout the day. Table 12 provides performance measurements when running three types of batch queries. I got different RDD sizes, because I ran the queries sequentially. The queries are sorted based on the order in which they were applied. This order helps to demonstrate the growth of the dataset from query to query.

<table>
<thead>
<tr>
<th>Query Type</th>
<th>RDDs (# Tweet)</th>
<th>Processing Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
<td>Max</td>
</tr>
<tr>
<td>Popular</td>
<td>495,328</td>
<td>527,288</td>
</tr>
<tr>
<td>Trend</td>
<td>824,103</td>
<td>973,009</td>
</tr>
<tr>
<td>Dataset</td>
<td>992,156</td>
<td>1,161,085</td>
</tr>
</tbody>
</table>

Table 12: Performance Measurement of Batch Queries

The results in Table 12 clearly show longer processing times for larger RDD sizes as expected. Generally, the measurements show good performance results for the batch queries. The highest average processing time is ~2.1 minutes for a million tweets, which is acceptable.

Figure 27 illustrates the processing times as the RDD size increases for these three queries. In each query, there is a slight increment in the batch processing time although the system faces growth of data overall. This steady increase of processing time as the RDD size increases shows some stability of the batch processing.
Figure 27: Batch Queries on US Presidential Election
5.2.3 Horizontal Scaling

Scalability is the ability to maintain performance under increased load of data received and processed by the system. Although the performance of the batch queries in the previous section showed stability even as the data size increased, I could improve system performance by adding more computing resources. The design of EPIC Real-Time platform is based on the powerful big data technologies of Kafka, Spark, and Cassandra. The horizontal scaling of each of these clusters would guarantee a significant difference in performance.

Our performance results were achieved when running EPIC Real-Time on my three-node cluster with each machine running Kafka, Spark, and Cassandra. I am satisfied with the performance results but confident they can be improved since each of these technologies provides horizontal scalability. To perform my evaluation, I allowed one Spark worker to process one query. If I allowed two workers or more to process a single query this would reduce the query processing time and provide a faster real-time response. Adding nodes to our cluster should improve performance in a number of ways: larger Twitter datasets can be collected; smaller analysis windows can be used; and more queries can be run simultaneously.

In addition, adding nodes to our cluster would allow us to experiment with different deployment strategies. One option (shown in Figure 28.a) is to simply add more nodes, with each node configured to run all three systems. With more nodes, I get more power and potential to reduce immediacy time. Another option (shown in Figure 28.b) is to add new nodes but then separate the systems. I could run Kafka on its own cluster and then run Spark and Cassandra on a second cluster. This might increase performance by reducing contention between the two systems.

It is worth noting that I do not want to separate Spark and Cassandra because I make use of the Spark-Cassandra Connector and it makes use of data locality to increase performance between these two systems. With data locality, we try to have Spark process just the data that is stored on its node rather than wasting time sending data stored in Cassandra on one node to be processed by Spark on a second node.

Unfortunately, I did not have access to the computational resources to investigate these potential configurations of my system. Regardless, my use of these technologies for EPIC Real-Time allows it to be configured for a wide range of data collection and analysis scenarios.
Figure 28: EPIC Real-Time Horizontal Scaling
CHAPTER 6

RELATED WORK

Much research has been invested in designing big data infrastructure, big data programming models like MapReduce, and developing data analysis techniques that can operate at scale. In this section, I present examples of this research that have particular relevance to my research.

The work in (Lei, et al., 2014) presented a big data infrastructure, called Redoop, that utilizes the MapReduce paradigm to support recurring queries in Hadoop. Recurring queries are those that are repeatedly executed for long periods of times on large datasets. Redoop employs window-aware techniques for such recurring workloads by including adaptive window-aware data partitioning, cache-aware task scheduling, and inter-window caching mechanisms. The authors argue that the Redoop infrastructure showed better performance than Hadoop for recurring workloads. The authors also argue that many extensions have been proposed on top of Hadoop, such as online processing and iterative queries, but none of those system support or optimize recurring big data queries. The Redoop infrastructure implements recurring queries over continuously updated historic data, while I implemented continuous queries over data streams.

Beyond Redoop, in the open source software community, there have been an extensive range of projects related to distributed streaming architectures for real-time processing. To highlight some examples, MapReduce Online supports continuous processing within and across different MapReduce jobs. S4, Impala, Storm, and Spark and Spark Streaming are general-purpose, distributed, scalable, fault-tolerant realtime computation systems (Liu, et al., 2014). Moreover, Twitter has open sourced their MapReduce streaming framework, called Summingbird, which integrates Hadoop and Storm as one of the first openly available Lambda Architecture compliant systems (InfoQ, 2014).

In fact, the emergence of cloud computing has provided cost-effective platforms to process high volumes of data. The work presented by (Kalashnikov, et al., 2015) developed a cloud-based data stream processing platform. The platform, called Cerrera, provides scientists with access to big data analysis tools to solve their scientific problems. Using a web interface, Cerrera allows
researchers to describe data stream processing workflows graphically, and then change them dynamically during the execution process for more interaction. Visualized results are displayed using different types of diagrams, plots, and tables. The data processing workflow makes use of a directed acyclic graph. Cerrera uses Apache ZooKeeper to manage its infrastructure and the work of all its subsystems. It also utilizes Kafka and Apache Storm for processing its data streams. The key features of the platform is dynamic parameter adjustment which allows users to customize data processing on-the-fly. The work on Cerrera is definitely related to my work. It provides researchers with capabilities to analyze data streams using a web interface by eliminating the need to deal with the complexity of the underlying infrastructure. It also allows researchers to interact with every process by changing it at runtime. However, the work does not provide Web services and APIs, as it is implemented as one codebase deployed in the cloud. I believe my service-based approach provides more flexibility to developers as well as extensibility to analysts.

Much research has been done to provide big data analytics as a service. Xu, et al. (2015) presented an architectural design of a real time data analytic service that allows machine learning models to be continually updated by real time data streams. The approach is to wrap the underlying big data processing framework as model training services and integrate them with data services and prediction services. The approach exposes all key elements in data analytics as reusable services in a service oriented architecture. The analytic processing is achieved by integrating batch and stream processing, using Spark as the big data framework and MLlib as a scalable machine learning library. The architecture is evaluated by using a runtime detection system driven by real-time logs and monitoring metrics. The authors argue that all available big data processing and analysis tools in the cloud are used only as software-as-a-service tools, while they focus on fine-grained RESTful service design. This work is similar to my work as its focus is on real-time analysis and reusability by providing big data analytics as reusable services; however, Xu’s work focuses only on providing dynamic model training services updated in real time. The work also leverages the capabilities of Spark as I do. My work is different in that I support a wider range of analysis and make use of REST services to implement a Lambda Architecture-based system and thus tackle batch processing concerns as well.

With more adoption of the Lambda Architecture, the work in (Villari, et al., 2014) addresses two main challenges associated with Internet of Things (IoT) scenarios; it looks at the management of large-scale smart environments using big data storage and analytics by presenting a software
solution inspired by the Lambda Architecture that provides real-time and batch data processing capabilities to an existing IoT system called AllJoyn. The software prototype presented in this work integrated MongoDB and Storm and was tested for analyzing a “smart home” case study. This paper provides an example of applying the Lambda Architecture for real-time and batch data processing to store and analyze big data in an IoT case study. This work leverages the Lambda Architecture design as I do, however, I follow different design and technology decisions and support a different problem domain. Some research has been performed to solve complexities associated with the Lambda Architecture. The authors in (Kroß, et al., 2015) solve the problem of redundant processing by presenting an analytical decision-making solution to develop on-demand stream processing. The authors in (Vanhove, et al., 2016) propose a solution to solve the complex synchronization introduced by the Lambda Architecture. They argue that the time when data is processed by the batch layer and the corresponding information needs to be removed from the speed layer introduces redundancy. In my work, my focus is to investigate how to provide the best performance of the Lambda Architecture to my end-user analysts.

Notably, analyzing social media data has become an emerging area of interest for both academia and industry. Twitter has been the most active social media platform connecting people by interchanging short messages (140-character tweets), in addition to media sharing. The literature is rich with research investigating the utility of Twitter data for a variety of concerns. Although there is a huge number of research papers dedicated to applying sentiment analysis, text mining, and natural language processing on Twitter data, I focus on comparing my work mainly with the research work of designing and implementing platforms for Twitter data analysis. Simmonds, et al. (2014) presented a platform for analyzing streaming and historical Twitter social data, solving the challenges of combining two systems in one while providing low latency query responses. First, the authors performed a meta-analysis of the type of questions that were been asked of Twitter data in three different domains: marketing and advertising, conferences, and emergency response. Their meta-analysis identified a small core set of generic queries that could answer several questions from various case studies. The paper presents the data model of the generic queries that reduces execution times and increase throughput by eliminating the cost of joins for the generic queries. Second, the paper presents the architecture of the platform that handles both type of continuous and traditional queries, by implementing a query monitor that directs queries either to a Cassandra database for historical Twitter datasets or to ESPER for querying Twitter
data as it streams in. Their evaluation of running different types of queries showed high response times, while reading from memory and disk. The cloud-based system achieves scalability by horizontally scaling the database across a set of cloud nodes. The meta-analysis presented in this work inspired my research to create my own set of generic queries based on my user study. The work presented in this paper is similar to my work in combining streaming and historical data analysis in one system, however, the architectural design is different than my Lambda Architecture based approach. The implementation of streaming and historical querying components is different, using Cassandra’s and ESPER’s APIs across a unified data model.

Project EPIC is not the only research group looking at efficient ways to analyze crisis data. Cameron, et al. (2012) presented an emergency situational awareness platform to help crisis coordinators detect incidents and identify event-related reports published by the public in Twitter. The platform, built using simple databases, a Java-based messaging system, and web services, provided a new source of data for an existing emergency management and crisis coordination system. Tin, et al. (2013) also presented an integrated framework for big data analytics to analyze disaster events. The framework developed a useful information prediction scheme based on co-occurring theory and Markov chain modeling to help improve the automatic detection of disasters via social media. The framework simulated results from data collected from social network platforms in the 2011 Japan earthquake and Tsunami. Moreover, González, et al. (2014) presented a tweet classification tool to detect people’s needs in natural disaster events. The work focused on improving the load balancing and reducing the processing time of stream processing tasks utilized by the Yahoo! S4 stream processing engine. My work represents another approach to analyzing crisis data by providing analysts with a set of queries to explore in real time.

Much big data research in the software engineering community has been devoted to making big data systems accessible to non-technical users. The work presented in (Malcolm, et al., 2014) adds accessibility to big data systems using a common services APIs, aiming for more adoption of big data technologies like Hadoop. Their paper proposes a solution that enables existing infrastructure to access big data systems via a services API while minimizing migration problems. Their proposed services API is platform and language independent but supports only Hadoop. Another effort in big data accessibility is presented in (Saleem, et al., 2014); it supports user exploration of large data sets by constructing ad hoc queries in a lightweight web-based framework called BigExcel for quick and easy analytics. BigExcel consists of three tiers for handling user
interactions with large data sets, constructing queries to explore data sets, and managing the underlying infrastructure. The communications between the three layers are based on RESTful services. The feasibility of this framework is demonstrated through two Yahoo Sandbox data sets for quantitative prediction and qualitative inference. Their work is relevant to my framework. It provides big data accessibility to researchers, allowing them to create ad hoc queries on datasets. However, my work makes use of REST services APIs, such that it is available to be consumed by both researchers and developers, using web interfaces and programming interfaces respectively.

An additional set of related work is devoted to microservice architectures. The paper presented by (Namiot & Sneps-Sneppe, 2014) provides an overview of microservice architectures and implementation patterns. It discusses the common principles behind microservices along with its advantages and disadvantages. This paper provides a good reference for my design and implementation of microservices in my own work. The work done by (Toffetti, et al., 2015) proposes a novel architecture that enables scalability and resilient self-management of microservices on a cloud infrastructure. The presented architecture is designed based on the concepts of service orchestration and distributed configuration management. To create resilient and scalable microservices, the approach utilizes distributed storage and leader election functionalities that are commonly available in current cloud architectures. The paper provides a good contribution in scalability of microservice architectures.

A final set of related work concerns end-user customization or generation of microservices. The authors in (Davies, et al., 2008) presented an infrastructure for the creation and sharing of end-user generated microservices. The paper provides a model showing the main elements of a microservice. The paper also presents a microservice life cycle showing the different stages in the life cycle of creating, deploying, and accessing microservices from the different viewpoints of a creator, a provider, or a user of a service. Davies also presented a research approach for a semantic infrastructure for user-generated mobile services via the concept of semantic microservices (Davies, 2009). The aim of this work is to enable easy creation, customization, and discovery of mobile services by end users. It contributes to three main areas: service description and modeling, knowledge warehousing and service discovery, and end-user service creation. The same group of authors also presented a novel design for user generated mobile microservices (Davies, et al., 2010). Mobile users can generate and authorize microservices based on an extensible model for the definition of a microservice. Functionalities for service creation and discovery are derived as
specified by the user. The authors argued this work will be an advancement towards a “semantically structured” internet of services.
CHAPTER 7

FUTURE WORK

A demanding research domain like crisis informatics will continually ask for more capabilities and features harnessing the power of data found in social media. The software engineering field needs to respond to the need of big data analytics reducing the gap between the state of the art of big data technologies and analysts. Moreover, the big data research area will continue to innovate around issues of high performance, fault tolerance, and scalability.

A lot of work can still be applied to EPIC Real-Time given its extensibility. I can extend the analysis capabilities by adding new types of queries. Adding a new query will require adding new Spark jobs (one each for the speed, batch, and service layers) to execute the query’s logic and create its result. It might also imply adding a new type of actor to the Query Engine, such that a new set of messages are designed to be handled by the new actor while being integrated with the existing actors already defined in the system. More analysis and data mining techniques can be integrated, allowing for additional Twitter analysis, such as processing geolocation information.

In the future, I aim to extend EPIC Real-Time to accommodate the analysis of any type of structured or unstructured data for fully exploiting the benefits of real-time big data analytics. For example, instead of collecting only Twitter data in Project EPIC, I can extend data collection and analytics to include data streams from other social media platforms like Facebook.

I need to add filtering features that can be applied on both streaming and historical datasets. A requested filter can be added to queries as a preprocessing step applied to the query’s tweet data source. I also need to add capabilities to export analysis results. Moreover, a lot of work still needs to be performed on the Web UI to provide a comprehensive analysis dashboard. As suggested by my analysts, I can visualize query results via graphs and charts. Another direction for future work is to integrate different types of third party tools, such as Tableau (http://www.tableau.com).

Furthermore, since the Lambda Architecture supports only one-way data flow (Piekos, 2015), I need to extend it to adopt one of the potential benefits of real-time analytics, namely real-time
decision-making. As shown in Figure 29, I can extend the Lambda Architecture by adding a new layer—the response layer—to create reactive event-oriented services. The response layer monitors all other layers to respond to a set of rules—predefined by analysts—that generate notifications when patterns are detected in the data. In the response layer, I need to implement complex event processing using Spark’s APIs. This layer would allow analysts to react to events in data streams automatically and increase their ability to respond to rapidly changing events.

Figure 29: A proposed extension to the Lambda Architecture
CHAPTER 8

CONCLUSIONS

Big data analytics refers to a wide range of practices and approaches to provide answers to expensive-to-compute queries on large datasets asked by analysts. The range of system types spans batch ETL systems to real-time analysis of streaming data. My new platform, EPIC Real-Time, aims to provide big data analytics via a core set of generic queries designed to support analysts in answering a wide range of questions while studying crisis events.

In this thesis, I present a lightweight, service-based platform called EPIC-Real-Time that allows crisis informatics researchers to explore Twitter data in real time as crisis events unfold. The main goal of the platform is to enable analysts to run different types of flexible and fast queries in real time and over historical data. The broader goal of the platform is to enable developers to build different big data applications using the platform’s RESTful APIs to create tailored analytical environments as needed by analysts. Although the new platform of EPIC-Real-Time is initially targeted to support crisis informatics researchers, I believe it can support real-time streaming and batch data analysis of a wide range of problem domains.

I presented the design and implementation of a system based on the core concepts of the Lambda Architecture that combine stream and batch processing in one system while providing many desirable features such as low-latency, fault-tolerance, extensibility, and generality. I used Spark and Spark Streaming for fast parallel processing; Cassandra for scalable and reliable data storage; and Kafka for Twitter data stream processing. I created my scalable message-driven application using the actor programming model as implemented by Akka. This helped me to build my service layer that provides accessible real-time and batch queries via Spark jobs, and allowing them to be used by my analysts. Akka enabled me to develop scalable distributed applications and resilient message-driven systems at a high level of abstraction.

EPIC Real-Time provides real-time streaming analytics in addition to batch analytics, solving the inherent complexities of these tasks while combining the best of both worlds. For crisis informatics, this research provides accessibility to next-generation big data streaming systems, like
Spark, while taking the first steps toward an integrated analytical environment. The real-time analytical environment allows analysts to ask different types of questions upon fast and massive amounts of streaming social media data while crisis events are still active, creating a new environment that is more capable than Project EPIC’s current batch-oriented systems. This makes the road for an analyst to valuable knowledge and wisdom faster and more efficient. For the software engineering field, the design and implementation of my real-time big data analytics platform validates many software development techniques and methods by bringing them together in one system and exploring what could be accomplished as a result. Specifically, recent development techniques in developing lightweight, maintainable, reactive, and distributed software services proved useful in creating EPIC Real-Time.


114


Villari, M., Celesti, A., Fazio, M., & Puliafito, A. (2014). AllJoyn Lambda: An architecture for the management of smart environments in IoT. In Smart Computing Workshops (SMARTCOMP Workshops), 2014 International Conference on (pp. 9-14). IEEE.


