

INCREMENTAL DATA COLLECTION & ANALYTICS
THE DESIGN OF NEXT-GENERATION CRISIS INFORMATICS SOFTWARE

by

AHMET ARIF AYDIN

B.S., FIRAT University, 2005

M.S., University of Colorado Denver, 2012

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:
Incremental Data Collection & Analytics
The Design of Next-Generation Crisis Informatics Software
written by Ahmet Arif AYDIN
has been approved for the Department of Computer Science

Professor Kenneth M. Anderson, Chair

Professor Judith Stafford

Professor Richard Han

Professor Qin Lv

Professor Gita Alaghband

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

AYDIN, Ahmet Arif (Ph.D., Computer Science)

Incremental Data Collection & Analytics The Design of Next-Generation Crisis Informatics Software

Thesis directed by Professor Kenneth M. Anderson

Everyday, enormous amounts of data are generated by a wide variety of computational systems. This data needs to be collected, stored, and analyzed to generate insights and information useful to the organizations performing this work. Typical workflows include consumer behavior interpretation, product recommendations, predicting future trends, and even support for emergency management before, during, and after mass emergency events. In the emergency management space, a new area of study—crisis informatics—examines how members of the public make use of social media during times of disaster. Crisis informatics software aims to collect and analyze the large amount of information generated on social media during times of mass emergency. In general, current crisis informatics software is focused on the batch processing of crisis data after an event has transitioned out of the immediate response and recovery phases. Now, there is a need to collect and analyze crisis data in real-time as it is streaming in during the crisis event itself.

This thesis offers an examination of the software architectures, techniques, frameworks, and middleware that are needed to augment crisis informatics software that make use of batch

processing techniques to perform data analysis with those that incrementally process, store, and analyze data as it arrives. This thesis work responds to the desires of analysts who need access to real-time data analytics and efficient batch data processing techniques to comprehensively analyze a mass emergency event. The techniques developed to achieve these goals have been implemented in a system called the Incremental Data Collection and Analytics Platform (IDCAP). This platform enables a comprehensive evaluation of the utility of these techniques. The system provides the following features: incremental data collection and indexing in real-time of social media data; support for real-time analytics at interactive speeds; highly concurrent batch data processing supported by a novel data model; and a front-end web client, known as the IDCA App, that allows an analyst to manage IDCAP resources, to monitor incoming data in real-time, and to provide an interface that allows incremental queries to be performed on top of large datasets.

DEDICATIONS

To

my parents and wife

who I will never forget their

patience, support and encouragements

ACKNOWLEDGEMENTS

I would like to thank all of the members of the EPIC Analyze project team—Mario Barrenechea, Mazin Hakeem, Adam Cardenas, and Sahar Jambi—as well as Jennings Anderson who have all supported my work. Also, I would like to thank all of my colleagues for their continuing support and friendship. Particularly, I sincerely would like to thank my advisor and mentor Prof. Kenneth M. Anderson. Thanks again for his amazing support since I have joined his research group at 2012. This thesis work could not have been completed without his support, encouragements, and effort. Finally, I would like to thank the directors of Project EPIC, Prof. Leysia Palen and Prof. Kenneth M. Anderson, who have provided an amazing environment to conduct my research.

This dissertation is based on work sponsored by the NSF under Grant IIS-0910586.

CONTENTS

Chapter

1. INTRODUCTION	1
2. BACKGROUND	3
2.1 Crisis Informatics & Project EPIC	3
2.2 Project EPIC Data Intensive Systems	4
2.2.1 EPIC Collect	4
2.2.1.1 Existing Cassandra Column Families	8
2.2.2 EPIC Analyze	11
2.2.3 EPIC Collect's Cassandra Column Family Design Issues	13
3. RESEARCH QUESTIONS & APPROACH	16
4. DATA INTENSIVE SYSTEM DESIGN CHALLENGES	19
5. INCRMENTAL DATA COLLECTION & ANALYTICS PLATFORM'S DESIGN AND ARCHITECTURE	24
5.1 Storage Layer	26
5.1.1 NoSQL (Cassandra) Column Family Design	27
5.1.1.1 Event_Tweets	28
5.1.1.2 Event_Information	31
5.2 Index Layer	32
5.2.1 Event_Abstractions	32

5.3 Service Layer	35
5.3.1 DataStax Enterprise	35
5.3.1.1 Spark Streaming	37
5.3.2 RabbitMQ	39
5.3.3 Redis	40
5.4 Application Layer	45
6. IDCA APP	46
6.1 Process & Event Manager	46
6.2 Real-Time Monitoring	48
6.3 Incremental Analytics	49
6.4 Discussion	53
7. EVALUATION	55
7.1 Evaluation of the Underlying Cassandra Column Families	55
7.1.1 Tweet Duplication	56
7.1.2 Tweet Distribution by Row keys	58
7.1.3 Row key Generation	55
7.2 Batch Data Processing	61
7.3 Application (System) Level Batch Processing	70
7.4 Discussion	72
8. RELATED WORK	75
8.1 Data Intensive Systems Design	75

8.2 Data Modeling for NoSQL Databases	79
9. FUTURE WORK.....	84
10. CONCLUSIONS	85
BIBLIOGRAPHY.....	89

TABLES

Table

5.1	First set of Redis data structures	42
5.2	Second set of Redis data structures.....	43
7.1	EPIC Collect events and indexing times.....	56
7.2	Tweet duplication count and size.....	57
7.3	Tweet distribution by row key	59
7.4	EPIC Collect row key generation time	60
7.5	The Queries used in the Evaluation	63
7.6	2012 Casa Grande Explosion query results	64
7.7	2013 Japan Earthquake query results.....	65
7.8	2013 Winter Storm Nemo query results	66
7.9	2013 Typhoon Philippines query results.....	67
7.10	2012 Hurricane Sandy query results	68
7.11	EPIC Collect Q9 vs Indexing in seconds.....	69
7.12	Query 10 Execution Results in seconds.....	71

FIGURES

Figure

2.1	EPIC Collect	5
2.2	The Software Architecture of EPIC Collect and EPIC Analyze.....	6
2.3	Filter_Tweet Column Family Design	9
2.4	Event_Filter Column Family Design	10
2.5	EPIC Analyze User Interface.....	12
5.1	The Software Architecture of IDCAP.....	25
5.2	The structure of the Event_Tweets Column Family	29
5.3	The structure of the Event_Information Column Family	31
5.4	The structure of the Event_Abstractions column family	33
5.5	DSE Cluster	36
5.6	DSE Spark Streaming	37
5.7	DSE Spark Cluster	38
5.8	RabbitMQ Admin Interface	40
6.1	IDCA App Process & Event Manager view	47
6.2	IDCA App Real-Time Monitoring tab view	48
6.3	IDCA App Real-Time Monitoring tab view	53
6.3.1	50
6.3.2	50

6.3.3.....	51
6.3.4.....	51
6.3.5.....	52
6.3.6.....	52
6.3.7.....	53
6.3.8.....	53
7.1 Query 9 Execution Results in seconds.....	69
7.2 Query 10 Execution Results in seconds.....	72

CHAPTER 1

INTRODUCTION

Each day, enormous amounts of data are generated by a wide variety of software systems: social media, web services, mobile apps, software simulations, sensor data, and the like. Between them, Google, Twitter, Facebook, and Microsoft capture, store, and process petabytes of data for various needs on a daily basis. This large volume of structured, semi-structured, and unstructured data is collectively referred to as *big data*. Big data is said to have five dimensions that need to be handled by developers creating software that operates on large datasets [Hu et al. 2014]. These dimensions are *volume* (the size of the data), *variety* (the different types of data from different resources), *velocity* (the data processing speed), *value* (the derived insight from the data) and *veracity* (the trustworthiness of the data). These dimensions make it clear that software engineers will encounter challenges when designing software systems to collect, store, and analyze large data sets [Anderson et al. 2015]. These data-intensive software systems need to be scalable, reliable, and resilient, able to store data in the presence of hardware- and software-related failures.

In 1971, Richard Hamming stated that “the purpose of computing is insight,” deriving information on top of raw numbers [Zhu et al. 2012]. As such, the purpose of collecting and storing massive amounts of data is to generate insights and information useful to the organizations performing this work. Furthermore, Michael Minelli, the co-author of the book *Big*

Data Big Analytics, states that “real-time big data is not just storing petabytes or exabytes of data in a data warehouse, but it is the ability to make better decisions and take meaningful actions at the right time and right place” [Barlow 2013]. These quotes provide the insight that storing large amounts of data to enable data analysis is an important and challenging task. Moreover, in [Jarr 2015], the author indicated that companies realized the importance of interacting with fast data (i.e. new data that streams in from a variety of data sources) to save their place in very competitive job market and stated that “data is fast before it is big.” He claims that “data in motion has equal or greater value than historical data,” and that companies need to adopt new approaches to handle fast data. Thus, providing analytics based on batch processing techniques and incremental real-time techniques are both very important to derive useful information out of large data sets. Typical workflows based on data analytics include consumer behavior interpretation, product recommendations, predicting needs or future trends, credit card fraud detection, as well as support for emergency management before, during, and after disaster events.

As a result, it is important to build tools that provide batch and real-time data analytics for analysts to manage, investigate, and analyze these data sets quickly and efficiently without losing a big picture view. To do this well, requires sophisticated approaches, techniques, and methods with carefully integrated data processing frameworks and technologies.

CHAPTER 2

BACKGROUND

This thesis work is performed in the context of Project EPIC, a large NSF-funded project, that investigates a wide range of crisis informatics topics. In this chapter, Project EPIC data intensive systems are presented to provide the context surrounding the work of this thesis.

2.1 Crisis Informatics & Project EPIC

Project EPIC (Empowering the Public with Information in Crisis) [Palen et al. 2009] performs work in a relatively new area of study—*crisis informatics*—that examines how members of the public make use of information and communication technologies (ICT)—such as social media—during times of mass emergency, and studies how their interactions are impacting emergency response and the way emergency management is performed [Palen et al. 2011]. Crisis informatics provides an opportunity to conduct interdisciplinary research drawing on a variety of technical backgrounds including software engineering, natural language processing, and human centered computing [Palen et al. 2010]. Crisis informatics introduces two main challenges for software engineers working to support this type of research: a) the need to collect/store large amounts of social media data during disaster events in real time without losing information and b) the need to manage and analyze the collected data efficiently.

2.2 Project EPIC Data Intensive Systems

Project EPIC has designed and developed two large-scale data intensive systems for collecting and analyzing Twitter data sets. These two systems are EPIC Collect [Anderson and Schram 2011; Schram and Anderson 2012] and EPIC Analyze [Anderson et al. 2015]. In this section, the capabilities of both systems and the features they provide to crisis informatics researchers are explained.

2.2.1 EPIC Collect

The primary capability needed to conduct crisis informatics research is large-scale data collection of social media data. The key challenge here is that social media data is ephemeral. Collecting Twitter data based on keywords while an event is occurring is needed because obtaining a complete set of data after the fact is nearly impossible for many organizations and cost prohibitive for large organizations.¹ Twitter provides two APIs to access public Twitter data: the Streaming API², and the REST API for acquiring tweets.³ The Streaming API allows tweets to be captured in near real-time based on a set of filters that are provided when the connection is first established. (They also provide a filtered stream of “all tweets”—known as the public stream—that are currently being submitted by Twitter users from around the world.) The main limitation to the Streaming API is that for high-frequency search terms, the number of tweets

¹ It is possible to purchase data sets from Twitter after the fact via the Powertrack Search tool. This application was originally developed by Gnip and then acquired by Twitter and is now part of their data products organization. Unfortunately, the cost of using such a tool to query the “Twitter firehose” is expensive and out-of-reach for most research groups.

² <https://dev.twitter.com/tags/streaming-api>

³ Twitter used to provide a third API---the Search API [The Search API 2016] --- but that functionality has since been merged into the REST API.

returned is filtered to be just one percent of the total amount of data available. The REST API provides generic access to a variety of Twitter objects including user profiles, user streams, twitter lists, and the like. The functionality of the old Search API was merged into the REST API recently and allows users to search for tweets that match a particular query on tweets that were generated within the last month. The same limitation discussed for the Streaming API applies to this functionality in that high frequency terms may only go back a few hours as opposed to the one month of data that is typically available for low-frequency terms.

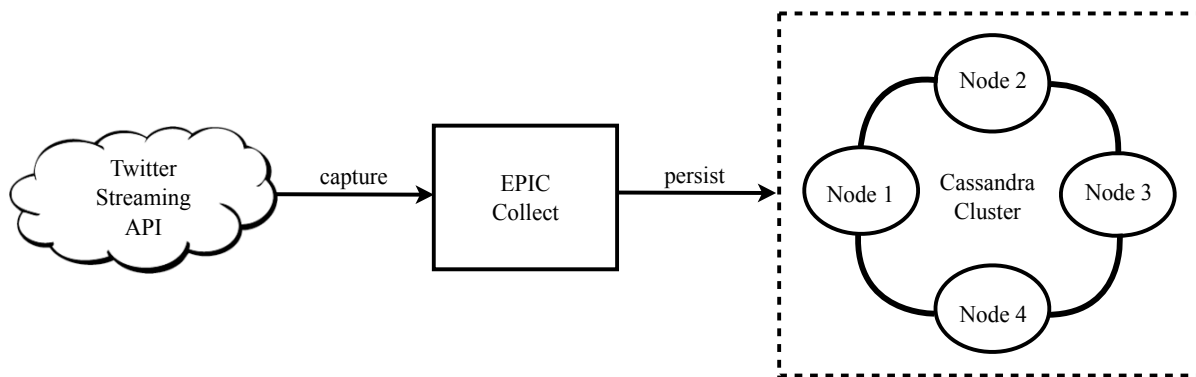


Figure 2.1: EPIC Collect

To conduct crisis informatics research, Project EPIC has been collecting and analyzing large Twitter datasets since 2009. Project EPIC's data collection software is known as EPIC Collect; it collects tweets from Twitter using the Streaming API. EPIC Collect is designed to be robust, reliable, and fault tolerant, and able to run 24/7 to collect data. Since 2009, approximately 2.5 billion tweets have been collected across hundreds of crisis events while maintaining 99% up-time. Tweets are stored in a 4-node Cassandra cluster to provide scalability and reliability and to ensure that it is nearly impossible to lose a tweet once it has been collected and persisted [Anderson and Schram 2011].

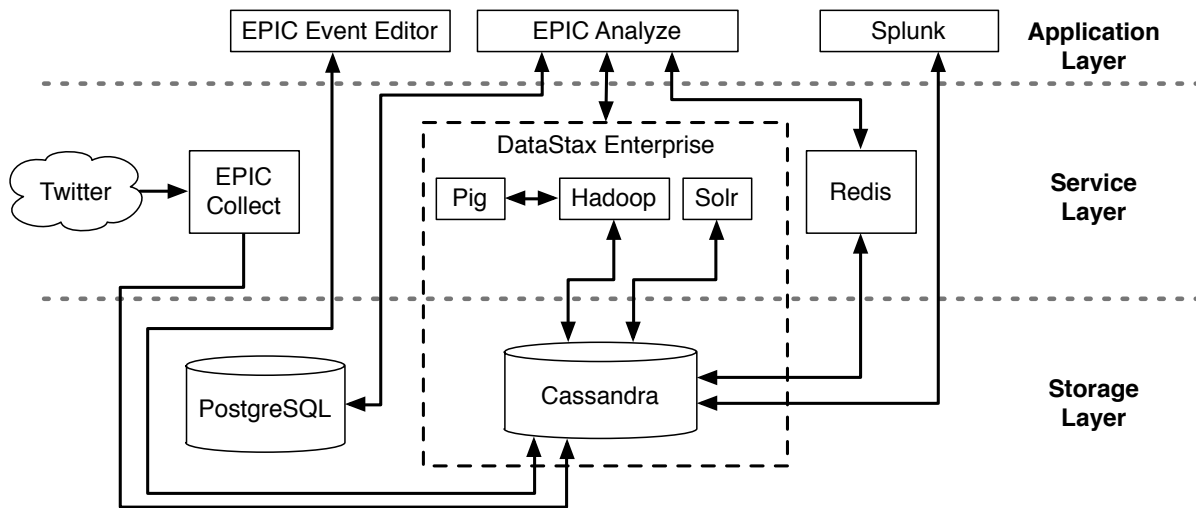


Figure 2.2: The Software Architecture of EPIC Collect and EPIC Analyze

While our storage requirements are several orders of magnitude smaller than Twitter and Facebook, we face significant challenges storing billions of tweets collected across hundreds of crisis events and processing those datasets. Project EPIC currently stores hundreds of crisis events in Cassandra; these events consume two terabytes of disk space. Each year, Project EPIC collects approximately fifty new events and those events, on average, will consume another terabyte of disk space. Detailed design information about EPIC Collect and the design of its architecture is explained in [Anderson and Schram 2011; Schram and Anderson 2012].

The application layer consists of three applications. The EPIC Event Editor is the user-facing application for EPIC Collect. It allows events to be created and maintained; it allows keywords to be associated with an event. Each time an edit is committed by the Event Editor to the Event_Filter column family, EPIC Collect disconnects from the Twitter Streaming API, reconstructs the set of active keywords from the updated status of the Event_Filter column

family, and reconnects to the Twitter Streaming API with the new set of keywords. This “reboot” of the connection takes on order just a few seconds and Twitter’s service is designed to “backfill” tweets that a Streaming API client missed while its connection was down (especially in the case of a few seconds worth of downtime). EPIC Analyze is a Ruby on Rails web application that accesses components in the service and storage layers to implement its functionality. Additional applications can populate this layer, such as third party data analysis applications. One such integration that Project EPIC performed was with the third party tool, Splunk, which is used to quickly get a feel for the data that is being collected for an event. Many data-intensive systems—like EPIC Analyze—live in an ecosystem of tools, with each tool providing services used for a specific purpose.

The service layer currently consists of EPIC Collect, components of DataStax Enterprise, and Redis. DataStax Enterprise (DSE) provides a collection of open source technologies that have been integrated to work with Cassandra.⁴ In particular, Apache Hadoop has been modified to read and write to Cassandra column families instead of HDFS. Apache Solr has also been modified to generate indexes based on information stored in Cassandra column families. Redis is a key-value in-memory database that organizes data into well-known data structures such as sets, lists, and dictionaries. EPIC Analyze makes use of Redis as a caching mechanism to achieve interactive speeds when browsing and paginating large Twitter data sets that consist of millions of tweets. The service layer, thus, is the extension point for integrating new technology into EPIC Analyze that allows it to implement its features and meet its goals.

⁴ <https://www.datastax.com/>

The storage layer consists of both NoSQL and RDBMS technologies, such as Cassandra and PostgreSQL. As discussed above, EPIC Collect stores Twitter data sets into Cassandra while PostgreSQL is used by EPIC Analyze to store information that it needs to track analyst information, affiliations, and admin-specific information [Anderson et al. 2015].

2.2.1.1 Existing Cassandra Column Families

EPIC Collect makes use of Cassandra, a widely-used NoSQL columnar database, to store tweets.⁵ Cassandra’s primary data representation is known as a *column family*. A column family is a collection of *rows* that are accessed by *row keys* [Lakshman and Malik 2010]. Each row is a collection of *columns* that consist of a *key* and a *value*. EPIC Collect stores tweets into a column family called Filter_Tweet (see Fig. 2.3). As discussed in [Anderson et al. 2015], Project EPIC stores tweets in this column family using a row key that has the following format:

`keyword:juliandate:tag`. The first portion of the row key refers to the keyword that was used to collect a tweet that is stored in this row. The second portion of the row key refers to the date tweets in this row were collected. The date is stored in Julian format; for instance, 2016001 refers to the first day of 2016. The final portion of the row key—referred to internally by Project EPIC as a ‘tag’—is a hexadecimal digit (0-9a-f); each tweet that is collected has an MD5 hash computed for it; the last digit of that hash is then used as the tag for that tweet.

⁵ <http://cassandra.apache.org/>

Filter_Tweet				
row key	tweet_id → JSON	...	tweet_id →JSON	...
japan earthquake:2013362:9	{ “created_at”: “Sat Dec 28 16:36:38 +0000 2013”..			
...				
fukushima:2013356:2	{“text”: They’re now trying to get people 55 & older to work on Fukushima reactors http://t.co/Hs4xkofwFt...			
fukushima:2013356:d	{“created_at”: “Wed Apr 25 09:21:14 +0000 2012”, “favourites_count”:3, “geo”: nil,...			
...				

Figure 2.3: Filter_Tweet Column Family Design

In this way, tweets containing a particular keyword collected on a particular day are distributed across at most sixteen rows in the Filter_Tweet column family. This approach is used to keep each individual row size down to a reasonable number of columns and also aids in the distribution of tweets across a cluster of machines. Once a row key has been determined for a particular tweet, it is stored as a column in that row. The tweet id used as the column name and the entire JSON representation of that tweet (as received from Twitter via the Streaming API) is stored as the column value. This row key design allows tweets to be written concurrently into the Filter_Tweet column family, provides ways to get tweets that contain certain keywords, and allows tweets within a certain time range to be retrieved in an efficient and scalable manner [Anderson and Schram 2011; Schram and Anderson 2012].

EPIC Collect makes use of a second column family—the Event_Filter column family (see Fig. 2.4)—to keep track of an event’s keywords and the data collection time range of each

keyword. The row key for the Event_Filter column family is a unique event name. Each row can contain a different numbers of columns. Again, each column is a key-value pair with the key being a keyword (search term) and the value being a dictionary that contains information about the collection of that keyword, such as the date that data collection was started for that keyword for that particular event. As is shown in Fig.2.4, “2013 Japan Earthquake” and “2012 Hurricane Sandy” are event names that are used as row keys. One keyword for the 2012 Hurricane Sandy event is “DNSY” and its value contains information about the data collection for that keyword for that event [Schram and Anderson 2012].

Event_Filter				
row key	keyword → value	keyword → value	...	keyword → value
2013 Japan Earthquake	{“EQjapan” → {“enabled”: false, “createDate”:1382725164559, “type”: “track”, “modifiedDate”:1402295000575}, "fukushima" → {Value} , “honshu japan” → {Value} ,}			
2012 Hurricane Sandy	{ “sandy” → { “enabled” : false, “createDate”:1351185745680, “type”: “track”, “modifiedDate”:1365197714404}, “DSNY” → {Value} , “superstorm” → {Value}, ... }			
...				

Figure 2.4: Event_Filter Column Family Design

These two column families form the basis for EPIC Collect’s ability to reliably store large amounts of Twitter data and then access that information for later analysis. Analysis currently consists of a large amount of batch processing applied to events that are no longer actively collected. Some of this work is performed by custom scripts that are invoked when the analysis of a particular event begins. Exploratory analysis is performed in EPIC Analyze which

is discussed next; additional analysis occurs after an analyst has used EPIC Analyze to discover a “working set” of tweets; these tweets are typically exported outside of EPIC Analyze and further manipulated by 3rd party tools such as Excel and Tableau. This situation is changing, however, as new features are added to EPIC Analyze that allow for additional computations to be performed and as additional visualization capabilities are added [Anderson et al. 2015].

2.2.2 EPIC Analyze

Project EPIC’s data analytics software is EPIC Analyze [Anderson et al. 2015]. Before the development of EPIC Analyze, Project EPIC analysts used custom in-house tools such as eDataViewer [Starbird et al. 2010], hand-written scripts, and third-party analysis tools. These tools were, in general, not designed to view/process large data sets. To make use of Excel and eDataViewer, Project EPIC’s datasets had to be reduced by several orders of magnitude. Before EPIC Analyze, analysts were performing tasks such as filtering, searching, and sampling in an *ad hoc* manner or with significant developer support. Also, analysts were performing searches on Twitter using third-party tools such as TweetDeck to monitor data sets as they were being collected, and analysts were performing searches on Twitter to investigate and visualize data sets during their analysis tasks [McTaggart 2012]. As a result, Project EPIC undertook the design and engineering challenge of producing a data analysis environment that could more directly support the analysis tasks of Project EPIC researchers. The first version of that environment, EPIC Analyze, is in active use and still under development; it adopts a batch-oriented approach to processing large social media datasets. Additional analysis occurs after an analyst has used EPIC Analyze to discover a “working set” of tweets; these tweets are typically exported outside of EPIC Analyze and further manipulated by tools such as Excel and Tableau. This situation is changing, however, as new features are added to EPIC Analyze that allow for additional

computations to be performed and as additional visualization capabilities are added. The design of EPIC Analyze and the details of its data model are explained in [Anderson et al. 2015].

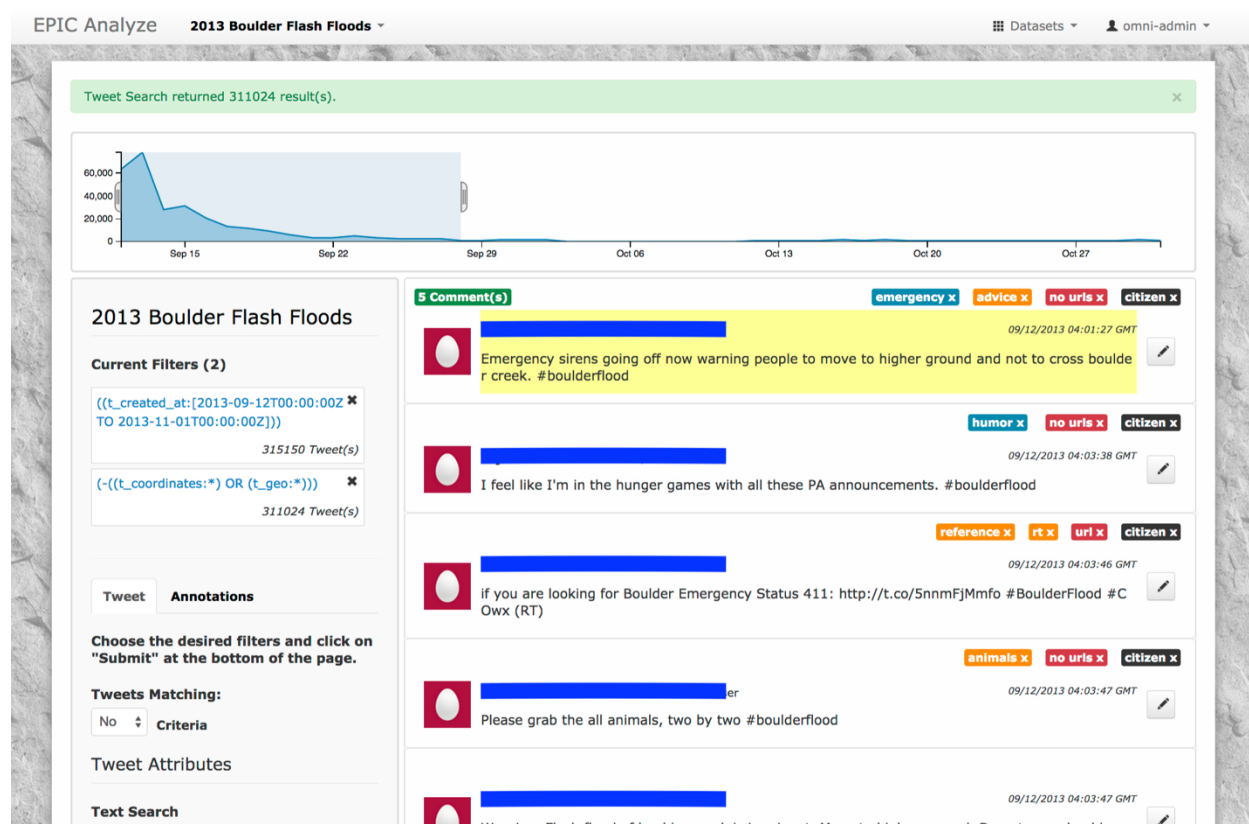


Figure 2.5: EPIC Analyze User Interface

The key design goal of EPIC Analyze is to provide scalable, extensible, and efficient software for performing crisis informatics research. The basic approach has been to batch process data sets ahead of time to generate indexes that allow a wide range of analysis tasks to be performed at interactive speeds. EPIC Analyze is a web application (see Fig.2.5) that is built on top of EPIC Collect's data persistence technology (Cassandra) and adds additional software frameworks (such as Hadoop, Solr, and Redis) to provide the capabilities required for ad hoc data analysis.

EPIC Analyze was developed based on a user-centered design process and its features were elicited based on crisis informatics researcher's requests. During the design process, agile-based methods were used to co-construct the platform alongside feedback from actual analyst end-users [Anderson et al. 2015].

The primary goal of EPIC Analyze's software architecture (see Fig. 2.2) is to achieve scalability and performance. This heterogeneous architecture is classified into three layers: applications, services, and storage. Our experience indicates that architectural heterogeneity is inevitable in the design of data intensive systems. The design of EPIC Analyze and the details of its data model are explained in [Anderson et al. 2015].

2.2.3 EPIC Collect's Cassandra Column Family Design Issues

Since 2009, EPIC Collect supported the creation of over 70 research publications on crisis informatics. Since that time EPIC Collect's data model (column families in Cassandra) have been perfectly storing Twitter data sets in a scalable fashion. However, there is still room for improvement with respect to their design. During the development of EPIC Collect, the focus was not on supporting data analysis but rather on supporting scalable and reliable data storage. Since 2013, I have been working on the EPIC Analyze project that is built on the top of the EPIC Collect. During this time, I have identified issues with the current design of the infrastructure that hinder its use for batch data processing and real-time data analysis. In this section, I highlight some of these issues and why they prevent real-time data collection and analysis and why they compromise efficiency of batch data processing and exploratory analytics. Many of the issues that I discuss are related to the current structure of the column families that EPIC Collect uses to store tweets. The design of these column families were discussed in Section 2.2.1.1.

The first set of issues are related to aspects of Cassandra's design that make it difficult to maintain a "big picture" view of the data stored within it. For instance, since Cassandra is able to create column families that contain millions of row keys, there is no operation that will return a list of row keys for a given column family in one operation. Instead, given that row keys are ordered (i.e. sorted), one has to ask Cassandra for the first "batch" of row keys and then for the next batch, and the next, until one has seen all of the rows that have been stored in a particular column family. For each batch, you save the last row key and use that as the starting point for the next batch. Your software thus has to have special logic to make sure it does not process one of these "boundary row keys" twice as it iterates through the list. While this aspect of Cassandra is unfortunate, it is the price of scalability. Cassandra allows you to create column families with a set of rows bounded only by available disk space; the price is that it does not attempt to keep a global view of all row keys that can be retrieved via a single API call.

Instead, we have found that it is better to adopt a row key design where such iteration is not required and instead generate row keys based on elements of your application domain. This is what EPIC Collect currently does. It does not have to keep track of all row keys; instead you ask it to give you all tweets that contain a particular keyword for a given set of days. From that information, you can easily reconstruct the row keys that are needed to retrieve those tweets.

Furthermore, everything discussed in this section about row keys applies to columns in a given row. That is, there is no way to ask Cassandra for the number of columns in a given row (for EPIC Collect, this corresponds to the number of tweets that have been stored in that row). Again, this is because Cassandra makes it possible to store millions of columns with a given row (bounded only by available disk space). As with row keys, column names are stored in an ordered fashion. As such, you once again must ask Cassandra for the first batch of columns for a

given row, and then the next batch, and so on until you have seen all of the columns. For each batch after the first, you supply the last column name from the previous batch to specify which batch of columns you are interested in. Once again, your software must have special logic to avoid processing a column twice as it iterates over all of the columns.

As a result, if you want to know how many tweets were collected for a given keyword on a given day, you must retrieve all of the row keys generated for that combination (recall that EPIC Collect may store up to 16 row keys per keyword/date combination due to its use of tags to distribute tweets across the nodes of a cluster) and then iterate over all of the columns in each of those rows.

A final issue with EPIC Collect's approach to storing tweets in the Filter_Tweet column family is that a tweet that contains multiple event-related keywords in it will be stored multiple times, once for each keyword. This leads to duplication that can cause difficulties down the line when computing metrics over the data set. For instance, identifying the most influential tweet across a set of keywords requires logic to avoid processing duplicate tweets, so as not to assign undue influence to a tweet that contains multiple keywords.

All of these issues will be addressed as a result of my research. In particular, I have developed a new design for these column families that eliminate these difficulties and lay the foundation for more efficient batch processing and the enabling of incremental, real-time analytics on Twitter data sets as tweets stream in during an event.

CHAPTER 3

RESEARCH QUESTIONS & APPROACH

Crisis informatics focuses on how to collect and analyze the large amount of information generated on social media during times of mass emergency. In general, Project EPIC's current crisis informatics software is focused on the batch processing of crisis data after its associated crisis event has transitioned out of the immediate response and recovery phases. However, there is a need to be able to collect and analyze crisis data as it is streaming in during the crisis event itself.

This thesis offers an examination of examines the software architectures, techniques, data analytics frameworks, (or tools) and middleware that are needed to augment crisis informatics software that make use of away from batch processing techniques to perform data analysis to those that incrementally process, store, and analyze data as it arrives during mass emergency events. Additionally, this thesis work includes understanding the needs of crisis informatics analysts with respect to real-time and incremental data analysis, to provide the techniques, middleware, and software architectural patterns needed to prototype next generation crisis informatics software that addresses those needs, and evaluating the capabilities of this new approach by comparing it with the capabilities of existing crisis informatics software. This thesis work is based on the following research questions.

RQ1: *How must the software architecture of a data analytics platform be designed to enable highly concurrent and incremental real-time data collection & analysis on large social media data sets?*

RQ2: *How can a software data model that is designed to support incremental real-time data collection and analysis also support highly concurrent batch data processing of large social media data sets such that the system can respond to analyst queries at interactive speeds?*

RQ1 drives the investigation of software architectures and software architectural styles that can support next-generation data analysis for crisis informatics research. To answer this question, I have used cutting-edge technologies that best support the modeling, processing, and persisting of data in an incremental and concurrent fashion in the context of Project EPIC. This question lead to the creation of the architectural design of the Incremental Data Collection and Analytics Platform (IDCAP) that provides incremental real-time data collection and analysis in a highly concurrent fashion.

Even though the focus of this thesis is identifying techniques and software architectures that can provide real-time data analysis capabilities as mentioned in RQ1, we cannot fail to support the batch functionality currently supported by EPIC Collect and EPIC Analyze. Thus, RQ2 required me to focus on how to also support efficient batch data processing. Both of these questions led me to identify the new data model, column family structures, and processing techniques that allow Twitter datasets to be stored such that both approaches to analysis can be supported in an efficient and scalable manner.

In this thesis, a bottom-up approach was followed to answer the research questions above. First, I started with RQ2 to design a novel set of Cassandra column families for IDCAP;

this new design was designed a) to address the limitations with EPIC Collect's original column family design including tweet duplication, row key generation, and wide rows; b) to provide a scalable way to store Twitter data sets that can be accessed efficiently; c) to provide support for both batch and incremental data analysis; and d) to provide a big picture view of all data sets stored within them. After evaluating the design of the new set of column families, I moved to answer RQ1 to finalize a novel and well-designed software infrastructure for the IDCAP that makes use of various cutting-edge technologies to provide incremental real-time data collection and analysis. Additionally, an example app called the Incremental Data Collection and Analytics App (IDCA App) was developed to provide the following functionality: a) the ability to orchestrate the IDCAP to stream, persist, and monitor tweets; b) an Event Manager to create, initiate, update, and close events and their keywords; c) a Process Manager to manage access to Apache Spark's Streaming capabilities and to invoke my persistence scripts as needed; and d) a Real-time Monitoring process that provides metrics about active events such as tweet distribution by keywords or the most recent geotagged tweets.

CHAPTER 4

DATA-INTENSIVE SYSTEM DESIGN CHALLENGES

In this chapter, the challenges related to the design of data intensive systems are discussed. The forthcoming challenges were encountered and handled during the design and development stages of the IDCAP. The primary challenges are proper architectural design by using the right tools to handle velocity of streaming data in a 24/7, reliable, fault tolerant, and scalable fashion; second, incrementally indexing the data streams in real-time to provide real-time data analysis on the fly and creating an indexing schema for later batch analytics; and handling the volume of data by permanently storing incrementally indexed data in a scalable fashion into a database to support batch data processing and analysis.

The design of software architecture is paramount [Perry and Wolf 1992; Garlan and Shaw 1993] because it explicitly impacts not only non-functional operational properties such as performance, reliability, and the availability of a software system but also non-functional quality attributes of a system such as changeability, reusability, and maintainability [Mattsson et al. 2006; Garlan 2000]. The result of rapid data growth has triggered the demand for a scalable data analytics platform architecture and a flexible data processing infrastructure to handle diverse and various analysis requests in real-time and later in batch processing on large amount of data. The design of a big data analytics platform's software architecture can vary based on different domains, needs, and requirements while keeping with core architectural design principles. The

desired characteristics of a real-time data collection infrastructure are reliability, scalability, accessibility, and the ability to store data in a concurrent and distributed fashion. Engaging with the proper architectural style and making the right technology trade-offs are important and challenging tasks in the development of data intensive systems. The software architecture of a system provides a big picture view of a system's components, the way those components interact, and their responsibilities [Bosch 2004; Hinsman et al. 2009]. Therefore, using the right set of technologies allows one to meet the design goals of the system and provides efficient generation of metrics and statistics and fast exploratory analysis on large datasets [Anderson 2015].

Capturing data in real-time in a scalable and highly concurrent fashion without losing any information is a challenging task. The challenges include handling velocity of the streaming data by a data collection service that strives to capture data in a 24/7, reliable and robust fashion, incrementally indexing streaming data in real-time in a reliable and fault tolerant fashion, and storing the indexed data in a scalable fashion to a permanent database. There are two difficult challenges related to indexing data streams in real-time. The first is what data structure can be used to store the tweets such that a global index is created across one or more data sets that is capable of answering the types of questions that crisis informatics researchers ask. This structure must be one that allows for efficient queries but also efficient inserts of new information into the index to allow for real-time data analysis. The second challenge is how can such an index be generated in an incremental fashion? Most techniques that operate in real-time divide the streaming data into short windows of time (e.g. 30 seconds). The challenge is to generate a mini-index of all tweets received in the last window, update all active metrics/queries, and then merge this mini-index into the global index before the next window of time has to be processed.

Data modeling is an important and challenging task. It is important since it involves decisions about how data is going to be structured and that, in turn, determines what you can do with it afterwards. In the context of Project EPIC, data modeling work is particularly performed with respect to the storage of Twitter datasets. In [Anderson et al. 2013], challenges related to modeling and analyzing Twitter data are discussed. Typically, in batch processing, analysts have a predetermined set of questions that they want answered for a given data set. Batch processing techniques examine every item in the data set and calculate the answers to those questions. With Project EPIC, those questions include basic metrics—such as the number of retweets in the data set, the most retweeted tweet, the most active contributor to the data set—and more complex questions, such as which user in the data set had the most influence, did users collaborate during the event, and so on.

Indexing is a core aspect of data management. For instance, an index can indicate how many unique tweets are in a data set in constant time simply by reporting how many entries exist in the index. Furthermore, the number of tweets that contain a particular keyword can be calculated in constant time by counting the number of tweets that are associated with that keyword in the index. Therefore, indexing plays an important role in data modeling because it can provide a big picture view of a dataset by computing metrics based on the index [Aydin and Anderson 2015]. Also, proper indexing provides a basis for parallel data computation; allows queries to be performed efficiently; and impacts how batch data processing and analytics are performed. There are two difficult challenges related to properly indexing datasets in order to accomplish these goals. First, what attribute (or attributes) of a data set should be used to create an index? Second, how can such an index be generated in an incremental fashion? These questions need to be handled to enable fast and efficient data analytics.

With respect to batch processing, a software engineer should design their data model to allow for processing to happen in a highly concurrent fashion. One approach that supports this in Cassandra is distributing data across multiple rows in the same way that MapReduce [Dean and Ghemawat 2008] splits large files of data into lots of small ones that are distributed across a cluster of machines. Cassandra is configured to spread data across a cluster by row key; therefore, if we consciously adopt this approach to ensure that our data is distributed across as many rows as possible, we set up the ability to access that data in parallel by multiple threads or multiple clients making requests on our Cassandra cluster at once. To address the lack of a big picture view in Cassandra with respect to row keys and column names, we can design additional column families to explicitly keep track of this information for us. That approach requires one column family to keep track of the row keys and column names of a second column family. The first column family can then act as an index allowing the information in the second column family to be accessed much more quickly. We will demonstrate this technique below in Chapter 5.

Creating abstractions on the top of large datasets is another important aspect of data modeling [Fekete 2013]. Proper abstractions facilitate exploratory analytics by creating maps of datasets based on the most requested domain attributes. For instance, to sort a large dataset of tweets based on tweet attributes—such as tweet id, username, or language—requires a significant amount of time if done on demand; however, if you create a “sorted” abstraction on top of the underlying storage of the tweets—as accomplished in our work with the incremental sorting method [Aydin and Anderson 2015]—then it is possible to allow the sorting of large data sets to occur instantly (since the sort order is pre-computed) in response to the requests of analysts while they study the data set. Furthermore, this abstraction can be maintained in an incremental

fashion as new tweets are collected for the data set being analyzed. Our work on the incremental sorting method makes use of the technique mentioned above in which some column families are used to simply store data while others are used to index the data stored in the former.

Additionally, beyond the mentioned challenges, data intensive systems require well-designed user interfaces to facilitate access to large data sets and to allow users to search, filter, sort, query, and analyze that data [Anderson 2015]. All of these challenges are addressed in the design and development stages of IDCAP which is discussed in the next chapter.

CHAPTER 5

INCREMENTAL DATA COLLECTION & ANALYTICS PLATFORM'S DESIGN AND ARCHITECTURE

In this chapter, we present the design and implementation of the Incremental Data Collection and Analytics Platform (IDCAP), including the techniques and technologies that were combined to achieve its functional and nonfunctional goals.

As discussed in Chapter 2, Project EPIC's existing data collection software—EPIC Collect—was not designed to provide rich support for data analytics. The goal of my work is to address this concern via the creation of a new data model, discussed below, and a new set of software services that implement the IDCAP. The IDCAP will not only be able to handle the scalable and reliable storage of streaming Twitter data (as EPIC Collect does now) but also provide the ability to perform real-time data analysis on incoming tweets as well as greatly improved and significantly faster batch data analysis on previously collected Twitter data set. While I focus exclusively on Twitter data due to the historical focus of Project EPIC's work, my techniques are easily generalizable to other problem domains and other types of data. The IDCAP is an example of a data-intensive software system; as a result, my work also provides insight into the types of techniques and technologies that must be combined to implement such systems and ensure that they are scalable, reliable, and efficient.

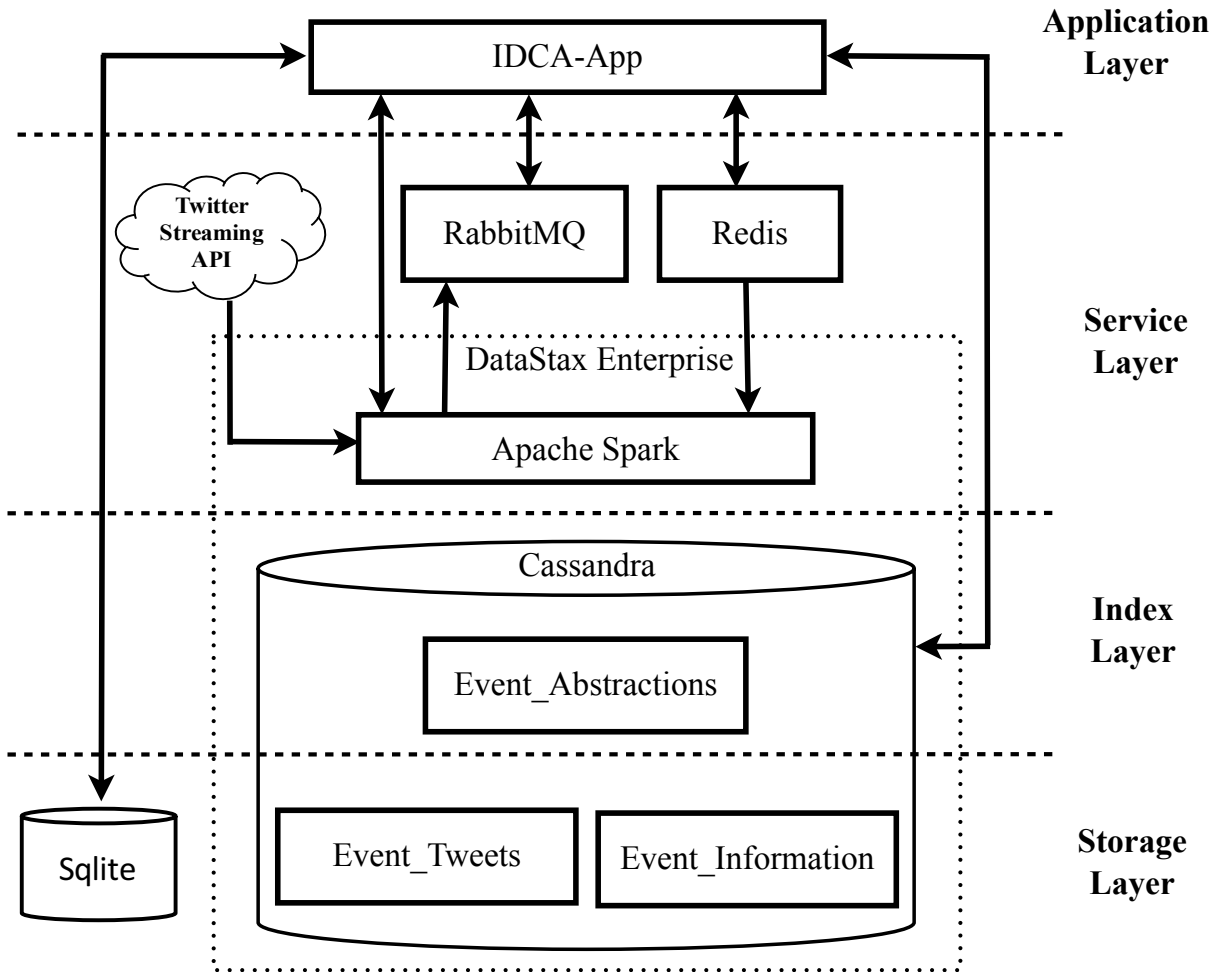


Figure 5.1: The Software Architecture of IDCAP

The software architecture of the IDCAP is presented in Figure 5.1. We adopt the layered architectural style to illustrate the high-level interactions of the components that comprise the IDCAP. The layered architecture style allows me to group technologies based on the role they play in the overall design. The architecture consists of four layers: application, service, index, and storage. Each layer is discussed next in a bottom-up fashion, starting with the storage layer.

5.1 Storage Layer

The storage layer consists of both NoSQL and RDBMS technologies; in the prototype implementation of the IDCAP, I specifically make use of Cassandra and Sqlite. Cassandra is used to store the Twitter data sets while Sqlite is used to store information by IDCAP's front-end web app to track user-specific information.

One of the key requirements of conducting research in crisis informatics research is to make use of the right database to achieve reliable, scalable, and accessible data storage without losing any data. Generic frameworks can be applied to solve problems across various domains. However, when a framework is not properly applied, an environment may not be able to meet its desired functionalities. Therefore, choosing the right technology is key to providing a desired set of features. To find the right database to meet Project EPIC's needs, various relational and NoSQL database technologies were explored including, MySQL [MySQL 2016], MongoDB [MongoDB 2016], HBase [HBase 2016], Solr/Lucene [Apache Solr 2016], and Cassandra [Apache Cassandra 2016]. Each of these persistence technologies has their own specialties. Cassandra was selected because of the following reasons. (1) Cassandra replaces the need for vertical scaling and sharding with its support for horizontal scaling, (2) Cassandra automatically partitions data across a cluster, eliminating the sharding issues encountered when trying to scale relational databases, (3) and Cassandra provides increased reliability due to its automatic support for replication [Anderson 2015]. It was for these reasons that the IDCAP also makes use of Cassandra to store Twitter data sets.

The IDCAP makes use of Cassandra's default partitioning strategy—the Murmur3Partitioner—to uniformly distribute data across a cluster based on MurmurHash hash

values.⁶ This strategy was chosen to uniformly distribute data across our Cassandra cluster and to avoid any performance problems that can be introduced by the use of Cassandra's random partitioning strategy. Moreover, to provide availability and accessibility, the replication factor for IDCAP for our three node Cassandra cluster was set to 3. This setting tells Cassandra to ensure that each of our cluster's nodes has a complete copy of the data stored in our data sets; this also ensures that our cluster can continue to respond to client requests even if two of the three nodes are down.

Finding the right data model for a given problem domain is critical to achieve fast and efficient queries. One way of achieving this goal is to carefully design the structures used by a given persistence technology; for relational databases, this refers to table design; for document-oriented NoSQL data stores, this refers to the structure of the documents, and for columnar NoSQL data stores, this refers to the design of the column families used to store and analyze data [Anderson 2015; Aydin and Anderson 2015].

5.1.1 NoSQL (Cassandra) Column Family Design

In this section, the design details of my new Cassandra column families are presented. As explained above, the new design avoids the limitations of the original EPIC Collect design that was focused on data collection not data analysis. The new design makes use of the techniques discussed above to work around some of the challenges associated with performing data analysis using Cassandra. The new design also eliminates the need to store tweets more than once; a given tweet is instead stored just once per event, and then indexes are created that allow that tweet to be discovered by any of its associated keywords. Since different events may have

⁶ https://docs.datastax.com/en/cassandra/2.1/cassandra/architecture/architecturePartitionerAbout_c.html

overlapping keywords, the same tweet may be collected for multiple events, if so, the tweet is stored more than once to allow the indexes for each event to be able to find that tweet without worrying about tweets that were collected for a different event. While duplication can still occur across events, such tweets are rare in our more than five years of performing this type of work, while the savings from not duplicating a tweet within an event because it contains multiple keywords is significant.

The new design consists of three new column families: `Event_Tweets`, `Event_Information`, and `Event_Abstractions`. During the design, development, and deployment of these column families, many decisions were made: to achieve scalability; to provide more compact storage of tweets; to create evenly distributed partitions of our data sets across a Cassandra cluster; to ensure that all tweets are stored sorted by the time they were created; and to simplify access to the tweets in a data set. All of these goals were achieved while ensuring that batch processing of tweets by EPIC Analyze occurs more quickly while carefully using storage and computing resources. Since `Event_Abstractions` stores indexes of data sets, it is discussed in the section on the index layer below (see Section 5.2). The `Event_Tweets` and `Event_Information` column families are explained next.

5.1.1.1 Event_Tweets

The `Event_Tweets` column family is designed to store tweets (specifically the JSON object that represents a tweet that Twitter provides via its Search, REST, and Streaming APIs) in a scalable and incremental fashion. Following our recommendation above, the row key and column names of this column family draw on elements of our application domain. In particular, the row keys of this table reference the name of a crisis event and are then tagged with a numerical suffix that starts at 1 and increases until all tweets for an event have been stored (more

details on this decision are presented below). The column names for this table are simply the tweet id of the stored tweet. Twitter already ensures that this id is globally unique and since we sort the columns by this id, our tweets come pre-sorted by the creation time of the tweet. In addition, if data collection ever goes down and we need to perform work to locate the tweets that we missed (by, e.g., making use of Twitter’s PowerTrack service to search back in time for tweets that were created while our system was down), when we have those missing tweets, we are guaranteed to be able to insert them into this table in sorted order without impacting any of the previously stored tweets. The Event_Tweets table is used to replace EPIC Collect’s Filter_Tweet table and avoids its need to store a tweet more than once due to matching multiple keywords as discussed above.

Event_Tweets					
row key	<i>tid 1 → Tweet JSON tid 2 → Tweet JSON tid 3 → Tweet JSON ... tid n → Tweet JSON</i>				
event_name:1	tid	tid	tid	...	tid
	Tweet JSON	Tweet JSON	Tweet JSON	...	Tweet JSON
event_name:2	tid	tid	tid	...	tid
	Tweet JSON	Tweet JSON	Tweet JSON		Tweet JSON
event_name:3	tid	tid	tid	...	tid
	Tweet JSON	Tweet JSON	Tweet JSON		Tweet JSON
...
event_name:k	tid	tid	tid	...	tid
	Tweet JSON	Tweet JSON	Tweet JSON		Tweet JSON

Figure 5.2: The structure of the Event_Tweets Column Family

In Figure 5.2, the design of Event_Tweets is shown. The row keys of Event_Tweets make use of an “event_name:k” pattern where event_name is the unique name of a data collection event. “k” is an integer that starts at 1 and increases incrementally. Each row contains

key value pairs where the “*key*” is the unique tweet id and the “*value*” is the tweet JSON object returned by Twitter’s various APIs. One design decision that we made concerns the number of tweets that are stored in a row. We decided to limit the maximum number of tweets in one row to twenty thousand. Thus, the first twenty thousand tweets are stored in `event_name:1`, the second twenty thousand in `event_name:2`, and so on. The only row that can have less than twenty thousand tweets is the last row, `event_name:k`.

This decision was driven by the maximum size recommended by DataStax—the company that drives the development of Cassandra—for a wide row. (In this context, a “wide row” simply means a row in Cassandra that has lots of columns.) DataStax recommends that wide rows be no larger than 100MB [DataStax 2016]. The reason for this is that when Cassandra stores and replicates data it does so at the level of row. If you store lots of data in a single row, then when Cassandra has to rebalance rows across a cluster (as happens when nodes are added or removed from an existing cluster), it has to spend a lot of time copying those large rows around and this work can have a negative impact on the performance of the cluster as a whole. Keeping the amount of data to be around 100MB, provides a nice trade-off between storing a good amount of data in a row and Cassandra’s ability to store and replicate rows while providing good performance.

To ensure that each row has, on average, only 100MB of data stored within it, we evaluated the average size of the JSON tweet objects that we have stored for several, previously collected events. We empirically determined the average byte size of a Tweet JSON object across three events containing millions of tweets; that size was ~4KB. We then used that number to determine how many tweet objects were needed to produce a row whose size equaled 100MB and that number was ~21.5K tweets. We then decided to pick twenty thousand as the maximum

number of tweets per row, ensuring that each row is less than the maximum size and therefore of a size which can easily be stored, replicated, and processed by Cassandra.

5.1.1.2 Event_Information

The Event_Information column family is designed to maintain summary information about the events. The row key for Event_Information is the name of an event. The columns of each row consist of standard set of columns that are present on each row and a set of columns that are unique to that row (see Fig. 5.3).

Event_Information								
row key	key 1 → value 1 key 2 → value 2 ... key n → value n							
event_name	keywords	status	julian_dates	et_rowkeys	kw1	kw2	kw3	...
	[kw1, kw2, kw3, ...]	{status →Active/Closed, event_process_date→Date, processed_rowkey_count → count}	[jd1, jd2, jd3, ...]	[event_name:1, event_name:2, event_name:3, ...]	{ is_active →“yes/no”, start_date → date_in_ms, end_date → date_in_ms }			
...								

Figure 5.3: The structure of the Event_Information column family

The four standardized column names that appear in each row are `keywords`, `status`, `julian_dates`, and `et_rowkeys` (which is short for “event tweet row keys”). Each keyword that is added to the event is stored in an array that serves as the value for the `keywords` column. The `status` column has as its value a map that tracks: the current state of the data collection; whether it is under active collection; when collection started or ended; and how many row keys have been created for that event. The `julian_dates` column tracks in an array all of the days that collection was active for the event. Finally, the value for `et_rowkeys` is an array of all row keys created for this event so far. These four columns provide a “*big picture*” view of an event and allow certain questions about the event (such as the total number of row keys or keywords) to be

looked up in constant time. Finally, for each keyword, there is a column that stores a map that tracks the current state of the data collection for that particular keyword: when was the keyword added; when was it removed; and is it currently under active collection.

5.2 Index Layer

Creating a proper mechanism to incrementally index large data sets while collecting data in real-time is an important and challenging task. Therefore, a well-designed indexing mechanism plays a key role in enabling fast data processing and analytics in real-time and provides benefits to the analysis of large data sets via batch processing as well.

The index layer provides an indexing schema on top of Project EPIC's large datasets via a novel design of the Event_Abstactions Cassandra column family. The Event_Abstactions column family provides indexes and high level abstractions for Twitter datasets that are stored in the Event_Tweets column family and it was designed based on the domain attributes that are most requested by Project EPIC analysts including unique event name, data collection date, tweet collection keywords, and unique tweet ids. Due to this new design, Project EPIC analysts will be able to filter, search, sample, sort, and perform exploratory analytics on large Twitter datasets. This analysis is currently supported in the IDCAP's front-end web app, IDCA App, and will eventually be integrated into a future version of EPIC Analyze. The IDCA App is discussed in Chapter 6.

5.2.1 Event_Abstactions

The Event_Abstactions column family is designed to create useful abstractions on top of the data sets stored in the Event_Tweets column family. These abstractions are metrics and other information that represent the most commonly requested attributes and/or questions about a data

set by Project EPIC analysts. These abstractions include an index to all tweets containing a particular keyword, an index of all geotagged tweets, and an index of all tweets collected on a given day. These indexes work by making reference to row keys and column names that exist inside of the Event_Tweets column family. That is, when performing analytics, our software will first consult the index in the Event_Abstactions column family and then retrieve the tweets that it references from the Event_Tweets column family. This is an example of the indexing technique discussed in Chapter 4 and it represents a way to bring more flexibility to Cassandra in terms of its ability to support data analysis.

The structure of the Event_Abstactions column family is shown in Figure 5.4. For each event stored in Event_Tweets, three row keys are created: “event_name:jd_keywords,” “event_name:jd_geotagged,” and “event_name:jd_index.”

Event_Abstactions					
row key	key 1 → value 1 (JSON) key 2 → value 2 ... key n → value n				
event_name:jd_keywords	event_name:1	event_name:2	event_name:3	...	event_name:n
	{jd1 → { kw 1→[tid's], kw 2→[tid's] ,..., kw n → [tid's] }, jd2 → { kw 1→[tid's], kw 2→[tid's] ,..., kw n → [tid's] }, ..., jdn → { kw 1→[tid's], kw 2→[tid's] ,..., kw n → [tid's] }}				
event_name:jd_geotagged	event_name:1	event_name:2	event_name:3	...	event_name:n
	{“jd1” →[tid's], “jd2” →[tid's], ..., “jdn”→[tid's]}				
event_name:jd_index	event_name:1	event_name:2	event_name:3	...	event_name:n
	{“jd1”→[tid's], “jd2”→[tid's], ..., “jdn”→[tid's]}				
*Column keys are Event_Tweets table row keys					

Figure 5.4: The structure of the Event_Abstactions column family

The column names for each of these rows are the row keys generated for the given event in the Event_Tweets column family. If an event “wildfire” had 150K tweets collected for it, then those tweets would be stored in the Event_Tweets column family across eight rows—wildfire:1, wildfire:2, ... wildfire:8—the first seven rows would have 20K tweets each and the last 10K tweets would be stored in the wildfire:8 row. The three rows in the Event_Abstractions column family for this event—wildfire:jd_keywords, wildfire:jd_geotagged, and wildfire:jd_index—would each contain eight columns with the names of the eight row keys from Event_Tweets.

The values for these columns across all three rows are hash tables. The keys for these hash tables are julian dates in the form YYYYDDD where the first day of 2016 is represented as 2016001 and the last day of 2016 is represented as 2016366 (since 2016 is a leap year). For the index of geotagged tweets and the index of tweets by day, the values of this hash table are simply arrays of tweet ids. The values for the index of tweets by keywords is different however. In that case, the value is another hash table in which the keys are keywords used in the data collection of that event and the values are arrays of tweet ids that contain those keywords.

With this structure, it then becomes straightforward to answer a range of questions about an event. If one wants to know how many tweets included the keyword *tornado*, then one iterates across all columns of the jd_keywords row for that event, then iterates across all days for each of the hash tables, and retrieves the array of tweet ids for that keyword and computes their size. This operation is significantly faster than what could be accomplished with EPIC Collect’s previous column family design. To do the same operation using just the Filter_Tweet column family would require long scans of multiple row keys and columns looking for tweets that contained the desired keyword and maintaining a global count while that scan was performed. In

actuality, we never answered questions like that using the old column family design. Instead, in EPIC Analyze, we would migrate tweets out of Cassandra and into Solr and then ask Solr to answer that question using its computed index. Now, with this new column family design, the need to use Solr to answer basic questions about an event is no longer needed and can be answered directly and efficiently by Cassandra itself. We still need Solr to generate an index to perform full-text search on the tweets that we collect but, that makes sense, that is what Solr was designed to do and there is no reason for us to attempt to duplicate that functionality.

5.3 Service Layer

The service layer consists of DataStax Enterprise [DataStax Enterprise 2016], RabbitMQ [RabbitMQ 2016], and Redis [Redis 2016]. Each technology and the purpose of use is explained next. Moreover, this layer is the extension point for integrating new technologies into IDCAP to implement new requests and to meet its future goals.

5.3.1 DataStax Enterprise

The IDCAP make use of DataStax Enterprise (DSE) since it provides a collection of open source Apache technologies that have been integrated to work with Cassandra; for instance, Spark [Apache Spark 2016] and Pig [Apache Pig 2016] have been modified to read from and write to Cassandra column families instead of the Hadoop Distributed File System (HDFS) [HDFS 2016]. Moreover, DSE's integrations allow multiple technologies to work together including Pig, Spark, and Solr.

For the prototype implementation of the IDCAP, a three node DSE Cluster was configured and deployed in an OpenStack environment (see Fig.5.5).⁷ Each DSE node is

⁷ <https://stack.cs.colorado.edu/>

configured as an OpenStack virtual machine that makes use of the CentOS operating system.⁸ Each OpenStack instance was configured to have its own extended volume to store data; each such volume was created to avoid data loss in the case of operating system crashes, problems with the openstack software, or hardware-related crashes.

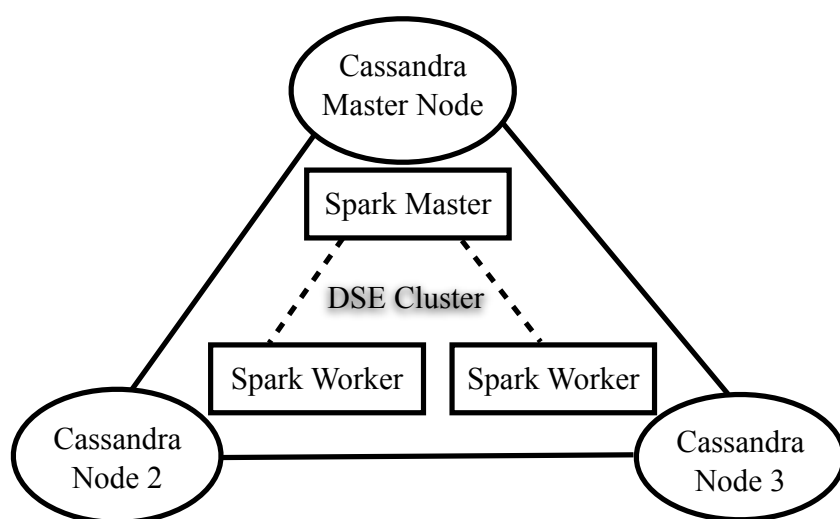


Figure 5.5: DSE Cluster

In the IDCAP's DSE cluster, DSE 4.7.3 is used; this version is deployed with Cassandra 2.1 and Spark 1.2.2.

Spark is a fast and general purpose cluster computing platform. Spark provides a simple way to parallelize applications across the clusters, and its API hides the complexity of distributed systems programming, network communication, and fault tolerance. Spark extends the MapReduce model in order to support more types of computations including interactive queries and stream processing. The Spark technology stack consists of Spark Core, Spark Streaming, Spark SQL, MLlib, and GraphX. Spark's philosophy of tight integration facilitates the use of

⁸ <https://www.centos.org/>

libraries and higher level components that combines different processing models for complex projects. Spark Core provides the functionalities of task scheduling, memory management, fault recovery, and interacting with storage systems. Spark core is the home for the RDD concept (Resilient Distributed Datasets) that represents a collection of items distributed across many server nodes that can be manipulated in parallel.⁹ Spark Streaming allows RDDs to be created from streaming data sources, such as Kafka, Flume, Twitter, ZeroMQ, Kinesis, or TCP sockets.

5.3.1.1 Spark Streaming

As mentioned above, Spark can ingest live data from multiple sources; it internally creates DStreams that consist of a series of RDDs and provides an API to transform/process each RDD that consists of data received from the source.¹⁰

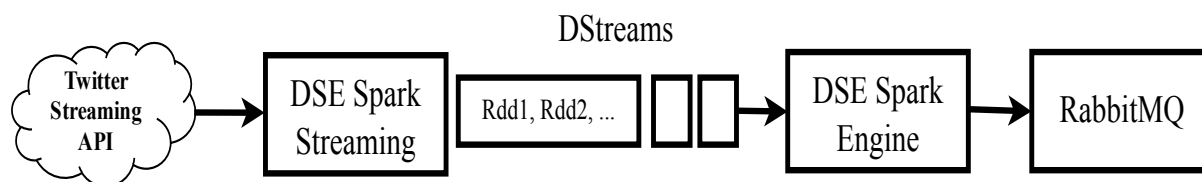


Figure 5.6: DSE Spark Streaming

In our case, the Spark streaming component is used to stream public tweets from Twitter (see Fig.5.6). To stream tweets using Twitter's Streaming API, the first required step is to have credentials for authentication that are auto generated by Twitter's developer tools after a Twitter app is created. After connecting to the Streaming API via those credentials, tweets can be streamed based on multiple parameters, such as keywords, users, language, or bounding boxes.¹¹

⁹ <http://spark.apache.org/docs/latest/programming-guide.html>

¹⁰ <https://spark.apache.org/docs/0.7.2/api/streaming/spark/streaming/DStream.html>

¹¹ <https://dev.twitter.com/streaming/overview/request-parameters>

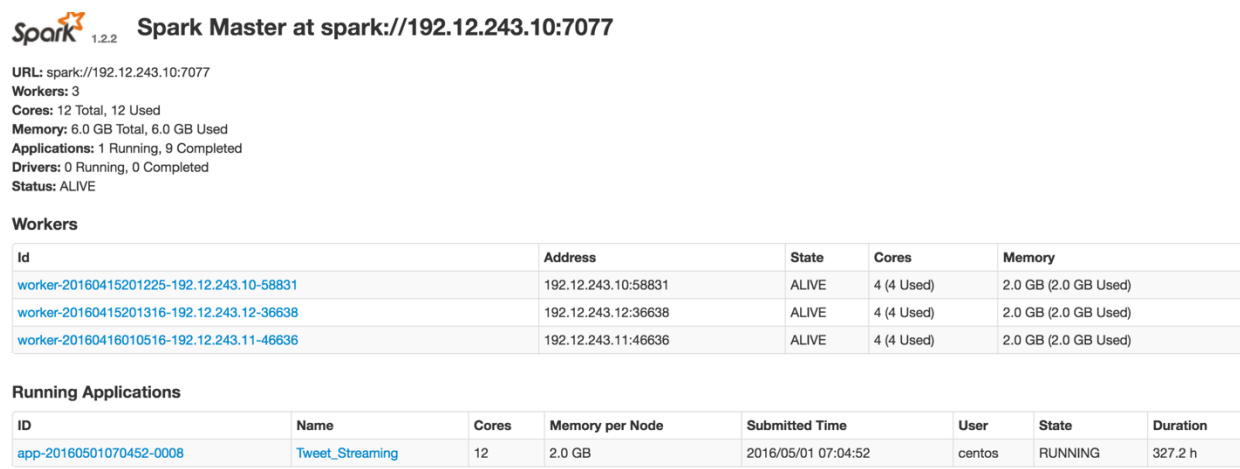


Figure 5.7: DSE Spark Cluster

Our spark streaming script is written in Python and makes use of the Pyspark module.¹² The streaming script performs its task via the following steps: 1) create a unique set of event keywords (limit is up to 400 keywords) by getting active event keywords stored in Redis (see Fig. 5.1); 2) creating a spark streaming context; 3) submitting keywords to the Twitter Streaming API via a POST HTTP request; 4) collecting RDDs that contain multiple Tweet JSON objects to the Spark master; 5) filtering out any responses that are not tweets; 6) classifying each of the remaining tweets based on the event keywords to place a particular tweet with an event(s) by checking the following tweet attributes: text, entities, hashtags, urls, quoted_status text, and retweeted_status text fields via the Python regular expression library; 7) creating RabbitMQ

¹² <http://spark.apache.org/docs/latest/api/python/pyspark.streaming.html>

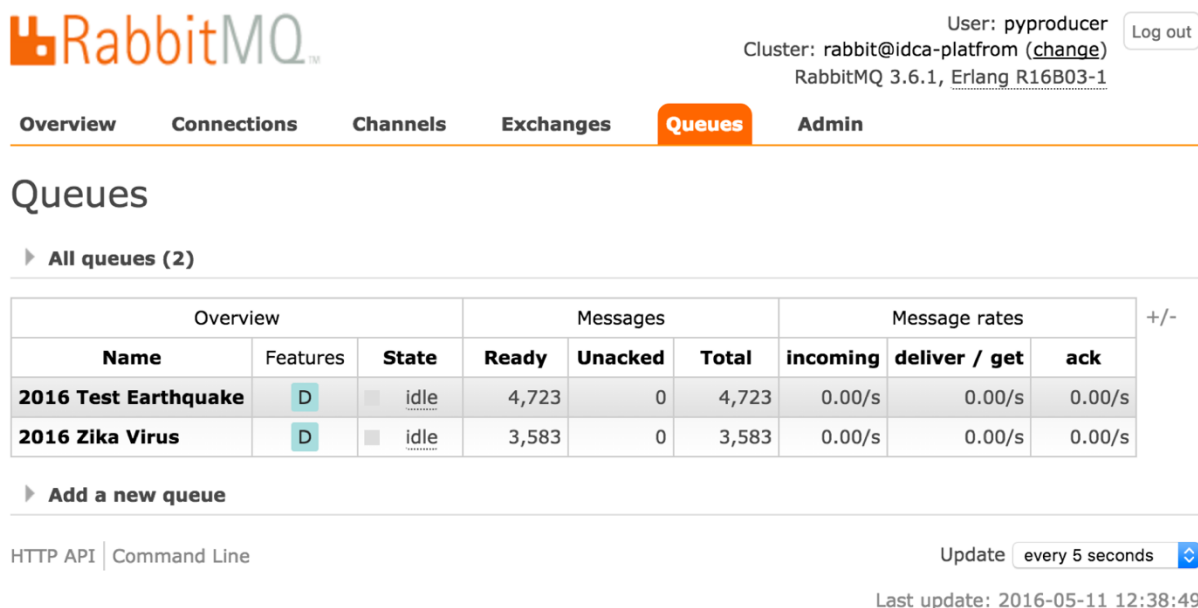
messages for each classified tweet; 8) inserting the messages created in step 7 into RabbitMQ queues; 9) and repeating steps 3 through 8 until a stop request is received.

I carefully configured my DSE Spark cluster and extensively tested the implementation of my streaming script to provide reliable, robust, efficient, and 24/7 Twitter data collection (see Fig.5.7). In particular, I configured some of the following parameters to help achieve the performance exhibited by the IDCAP prototype: 1) setting the batch interval time to 2 seconds; 2) setting the `Spark.streaming.unpersist`¹³ property to automatically delete a persisted stream from memory when they are not used anymore; this setting is needed since Spark DStreams are persisted in memory by default; and 3) setting `Spark.cleaner.ttl` property to 300 seconds to periodically clean memory since the default is “infinite” which is not conducive to 24/7 data collection.

5.3.2 RabbitMQ

RabbitMQ [RabbitMQ 2016] is an open source message queuing service that runs on all major operating systems and supports multiple programming languages to create reliable, durable, and persistent message queues. It provides an API that allows multiple clients concurrently to insert messages into queues and retrieve messages from queues.

¹³ <https://spark.apache.org/docs/1.2.0/streaming-programming-guide.html#memory-tuning>



User: pyproducer [Log out](#)

Cluster: rabbit@idca-platfrom ([change](#))

RabbitMQ 3.6.1, Erlang R16B03-1

[Overview](#) [Connections](#) [Channels](#) [Exchanges](#) **[Queues](#)** [Admin](#)

Queues

► All queues (2)

Overview			Messages			Message rates			+/-
Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack	
2016 Test Earthquake	D	idle	4,723	0	4,723	0.00/s	0.00/s	0.00/s	
2016 Zika Virus	D	idle	3,583	0	3,583	0.00/s	0.00/s	0.00/s	

► Add a new queue

HTTP API | [Command Line](#)

Update [↕](#)

Last update: 2016-05-11 12:38:49

Figure 5.8: RabbitMQ Admin Interface

The IDCAP makes use of RabbitMQ to provide a set of durable and persistent queues (see Fig. 5.8) that are used to receive tweets from the Twitter Streaming API via the work of the spark streaming script discussed above. For each event collected, the spark streaming script will create one queue in RabbitMQ; note, events are created/edited by the IDCA web app that will be discussed below in Chapter 6. These queues are used to temporarily store incoming tweets before they are permanently stored in Cassandra by the IDCAP persistence script. These queues thus serve as a buffer in a producer-consumer relationship between the streaming script and the persistence script allowing the two to work independently of each other and to shield each other from errors that might occur in the other.

5.3.3 Redis

Redis [Redis 2016] is a key-value in-memory database; it provides a rich API over a well-known set of data structures such as sets, lists, and dictionaries. Redis stores data in memory by default; on the other hand, Redis can also persist data permanently into disk as well

as memory by configuring its *appendonly* attribute from *no* to *yes* and specifying a location to store data on disk.

The IDCAP makes use of Redis to enable a wide range of its analytics capabilities including the ability to answer “big picture” questions about data sets under active collection, to allow it to incrementally index data as it arrives, and to support customizable queries on that indexed data in real-time. To provide reliable, accessible, and efficient real-time analytics and incremental analytics at interactive speeds, two set of Redis data structures are created.

The first set of Redis data structures is used to keep track of the current state of all data collection events as well as the status of the streaming and persistence scripts. The names of each data structure is listed in Table 5.1 along with a brief description of what information is stored in each one. These values are updated by actions performed in the IDCA App. For instance, `app_status_hset` is used by the IDCA App to orchestrate the streaming and persisting processes. Creating a new event by IDCA App triggers `event_name:summary_hset` to be created and `active_collection_set` to be updated. Adding new keyword(s) to an event by IDCA App triggers an update in `event_name:active_keywords_hset` or closing an event’s keyword(s) triggers update in both `event_name:active_keywords_hset` and `event_name:closed_keywords_hset`. Closing an active event triggers the event name to be deleted from `active_collection_set` and added into `closed_events_set`. Moreover, `streaming_pid_set` contains the active running process id of the spark streaming script in DSE Spark Cluster while `persisting_pid_set` contains the process id of the running persisting script and both structures are used with the purpose of eliminating zombie processes.

Table 5.1: First set of Redis data structures

Redis Data Structure	Purpose of Use
app_status_hset	Tracks the status of the streaming and persisting scripts
active_collection_set	Provides a unique set of active event names
closed_events_set	Provides a unique set of closed event names
streaming_pid_set	Stores the process id of the active spark streaming script
persisting_pid_set	Stores the process id of the active persisting script
event_name:summary_hset	Tracks the global state of an event
event_name:active_keywords_hset	Provides a map of active keywords and creation date
event_name:closed_keywords_hset	Provides a map of closed keywords and closing date
twitter_other_messages_set	Stores Twitter compliance messages received while streaming

Additionally, `event_name:summary_hset` provides the following information for a data collection event: status (active or closed), creation date, current active row in the `Event_Tweets` column family for this event, total tweet count, and current row tweet count.

The second set of Redis data structures listed in Table 5.2 are created for each active event to keep track of each event's information such as keywords, collection dates, indexes, and the JSON objects of collected tweets. These data structures and `event_name:summary_hset` are incrementally updated by our persisting script that is written in the Ruby programming language. This script concurrently consumes messages from the multiple queues created by the spark streaming script.

Table 5.2: Second set of Redis data structures

Redis Data Structure	Purpose of Use
event_name:tids_set:JD	Keeps a unique set of tweet ids
event_name:current_row_julian_date_set	Provides a unique set of julian dates for the current window (active row)
event_name:current_row_kw_set	Stores a unique set of keywords for the current window
event_name:analytics_tweets_hset	Provides a map of tweet id and tweet JSON for current window
event_name:ei:kw_set	Provides a unique set of event keywords for Event_Information
event_name:ei:jd_set	Provides a unique set of tweet collection dates in Julian date format for Event_Information
event_name:ei:et_rowkeys_set	Provides a unique set of Event_Tweets row keys for Event_Information
event_name:ea:geo_set:JD	Stores indexes of geo-tagged tweets for Event_Abstactions
event_name:ea:index_set:JD	stores indexes of all tweets by day for Event_Abstactions
event_name:ea:KW_set:JD	Provides keyword day indexes for Event_Abstactions

For example, the event_name:tids_set:JD tracks the unique ids of tweets seen in the current collection window for the given event name. The event_name:ea:KW_set:JD data structure represents multiple sets based on multiple event's keywords (KW) and tweet collection dates (JD). For example, assume a *Test* event collects on two keywords—*colorado* and *boulder* on two days—2016001 and 2016002—for the current collection window. Then, the following

four sets are created to keep track of the tweets that were collected on that day for that keyword for that event: `Test:ea:colorado_set:2016001`, `Test:ea:colorado_set:2016002`, `Test:ea:boulder_set:2016001`, and `Test:ea:boulder_set:2016002`. As new tweets come in 2016002, they will be assigned to the correct set by the persistence script.

To efficiently make use of my Redis data structures, defining the right “window size” is critical. This value determines the amount of data that will be stored in Redis (and therefore in memory) for each event. The window size has a direct impact on my goal of providing analytics in near real-time at interactive speeds for the analysts working with it. The window size I selected was the last 20 thousand tweets received from Twitter, which corresponds to the length that was selected for the size of rows in the `Event_Tweets` column family. This decision means that I’m never asking the persistence script to insert more tweets into Cassandra than its maximum recommended row size and this allows my system to remain responsive, efficiently making use of memory, disk, and network bandwidth. The persistence script first pulls tweets from RabbitMQ and places them in Redis and then when the window size is reached, flushes the tweets captured for the current window into the `Event_Tweets`, `Event_Abstractions`, and `Event_Information` column families.

Given the above information, the high-level responsibilities of the persistence script are now clear. For each active event, it subscribes to the corresponding queue in RabbitMQ; it gets notified by RabbitMQ when a tweet has arrived in the queue; it dequeues the tweet, checks to see if the tweet is unique (by consulting with the `event_name:tids_set` of the current julian day in Redis); it stores all unique tweets into a batch until 256 tweets are in the batch and then stores them into the current row of `Event_Tweets` for that event; it then updates the Redis data structures related to the `Event_Information` and `Event_Abstractions` column families, so they are

ready to respond to queries; it then loops and performs these steps again until the window size of 20 thousands tweets total has been reached, it then flushes the Event_Information and Event_Abstractions information to Cassandra. It continues to do this until it is told to shutdown by the IDCA App.

When an active event is closed, first, the IDCA App updates the event's data structures listed in Table 5.1; then the persistence script consumes all possible tweets of the event stored in RabbitMQ, updates all event related data structures listed in Table 5.2, performs a final update of the Event_Information and Event_Abstractions indexes of the event in Cassandra to permanently store the event's indexes, and the final steps is to delete the event's Redis-related data structures to release resources (i.e. memory) for future events.

5.4 Application Layer

The application layer contains the IDCA App that is the user-facing web application that provides analysts with access to the components of the IDCAP. The IDCA App provides a user-friendly UI that allows analysts to focus on their analysis tasks without having to worry about the complex orchestration of IDCAP components going on in the background.

CHAPTER 6

IDCA APP

The IDCA App, a Ruby on Rails web application, was developed to orchestrate the IDCAP and its resources. The IDCA App sits on top of the IDCAP architecture and provides the following features: a) allows analysts to efficiently create/update/close events and their keywords, b) orchestrates the spark streaming and persisting processes, c) allows analysts to monitor streaming tweets in real-time, and d) displays event metrics to analysts by providing customizable queries on the streaming data at interactive speeds. The IDCA App provides this functionality via a tabbed interface; the three primary tabs are called Process & Event Manager, Real-Time Monitoring, and Incremental Analytics. These tabs are discussed next.

6.1 Process & Event Manager

The Process & Event Manager (see Fig. 6.1) allows analysts to manage events with a primary focus of specifying their keywords. Creating a new event triggers the creation of the `event_name:summary_hset` data structure in Redis. Adding keywords to an event updates the related Redis data structures listed in Table 5.1. Furthermore, the IDCA App provides a process manager to manage the Spark Streaming and Persisting processes discussed in Chapter 5.

Process Manager

Process	Status	Action
Spark Streaming	Yes	<button>Stop Streaming</button>
Persisting to Cassandra	No	<button>Resume Persisting</button>

Active Events and Keywords Editor

Create a New Event

Event Name Create Streaming Event

Event Name	Add Keywords	Existing Keywords																		
2016 Zika Virus	<input type="text"/> <button>Add Keywords</button>	<table border="1"> <thead> <tr> <th>Keyword</th> <th>Creation Date</th> <th>Action</th> </tr> </thead> <tbody> <tr> <td>zika</td> <td>2016-04-22 00:53:40 -0600</td> <td><button>CLOSE</button></td> </tr> <tr> <td>#zika virus</td> <td>2016-04-22 00:53:40 -0600</td> <td><button>CLOSE</button></td> </tr> <tr> <td>#ZikaVirus</td> <td>2016-04-22 00:53:40 -0600</td> <td><button>CLOSE</button></td> </tr> <tr> <td>#ZikaChat</td> <td>2016-04-22 00:53:40 -0600</td> <td><button>CLOSE</button></td> </tr> <tr> <td>#Zikasummit</td> <td>2016-04-22 00:53:40 -0600</td> <td><button>CLOSE</button></td> </tr> </tbody> </table>	Keyword	Creation Date	Action	zika	2016-04-22 00:53:40 -0600	<button>CLOSE</button>	#zika virus	2016-04-22 00:53:40 -0600	<button>CLOSE</button>	#ZikaVirus	2016-04-22 00:53:40 -0600	<button>CLOSE</button>	#ZikaChat	2016-04-22 00:53:40 -0600	<button>CLOSE</button>	#Zikasummit	2016-04-22 00:53:40 -0600	<button>CLOSE</button>
Keyword	Creation Date	Action																		
zika	2016-04-22 00:53:40 -0600	<button>CLOSE</button>																		
#zika virus	2016-04-22 00:53:40 -0600	<button>CLOSE</button>																		
#ZikaVirus	2016-04-22 00:53:40 -0600	<button>CLOSE</button>																		
#ZikaChat	2016-04-22 00:53:40 -0600	<button>CLOSE</button>																		
#Zikasummit	2016-04-22 00:53:40 -0600	<button>CLOSE</button>																		

Figure 6.1: IDCA App Process & Event Manager view

When an analyst makes a change to an event, the Spark streaming process needs to be stopped and then restarted. Currently, this is done manually via the “Stop Streaming” button shown in Fig. 6.1. When this button is clicked, the spark streaming process is gracefully disconnected from Twitter and then stopped after all existing Twitter data has been deposited into the relevant RabbitMQ message queues. The user can then click on the “Resume Streaming” button to have the spark streaming process started again; it will then reconnect to Twitter using the updated event information and resume collecting data once again. This process is currently handled automatically by EPIC Collect and I intend to update the IDCAP to remove the need for manually starting/stopping the spark streaming process in response to event updates.

6.2 Real-Time Monitoring

The Real-Time Monitoring tab allows active events to be monitored in near real-time. The values presented in this tab's charts (see Fig.6.2) are calculated by indexes stored in Redis for the current window of each event. As shown in Fig. 6.2, three columns of information are provided for each active event.

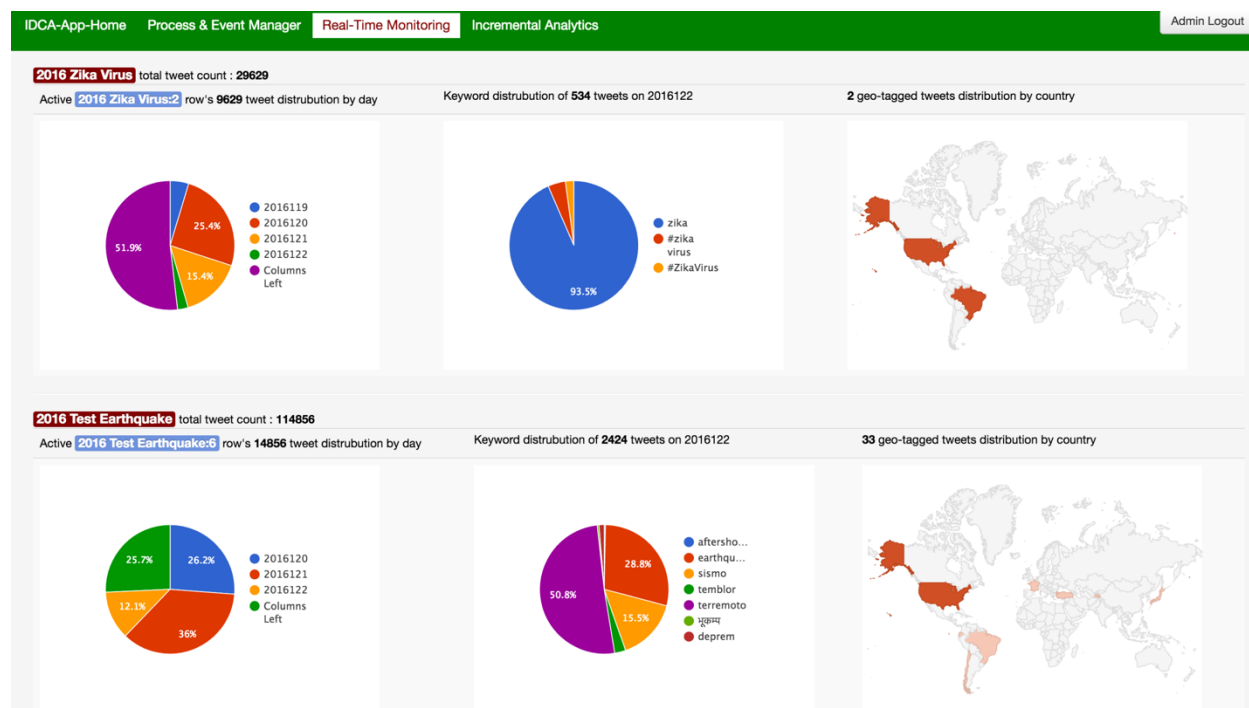


Figure 6.2: IDCA App Real-Time Monitoring tab view

The first column shows the tweet count of the event and the day distribution of the active row (a row of 20K tweets can easily contain tweets from multiple days) and it provides tweet distribution percentage by days that exist in the active row. Also, this column provides the number of columns left in the active row (i.e. the space for tweets before this row is full). As shown in Fig. 6.3, the active row of the 2016 Test Earthquake event is 2016 Test Earthquake:6 and it contains 14,856 tweets. This row contains tweets that were collected on the

following days: 2016120, 2016121 and 2016122. Also, 25.7 percent of columns are left before this row is considered full and a new active row is created.

The second column shows keyword distribution of tweets that are collected on the last day of the event in real-time. As shown in Fig. 6.3, 2,424 tweets have been collected on the last current day (2016122) of the 2016 Test Earthquake event and the tweet distribution chart is based on the seven keywords shown in that chart's legend.

The last column displays maps of the geo-tagged tweets that exist in the event's active row. As shown in Fig. 6.3, the 2016 Test Earthquake event contains 33 geo-tagged tweets that are stored in the 2016 Test Earthquake:6 row of the Event_Tweets column family. To create this map, all the geo-tagged tweets that exist in the current row are processed by making use of following Redis data structures:

- “2016 Test Earthquake:ea:geo_set:2016120”,
- “2016 Test Earthquake:ea:geo_set:2016121”,
- “2016 Test Earthquake:ea:geo_set:2016122”,
- “2016 Test Earthquake:analytics_tweets_hset.”

All of the columns in this tab are updated whenever the persistence script consumes tweets from RabbitMQ; this update occurs roughly at 15 second intervals which provides a near real-time feel for the monitoring.

6.3 Incremental Analytics

The Incremental Analytics tab provides analysts with a user interface to incrementally process queries on the entire set of tweets for an event including all tweets stored for the event in Cassandra as well as all tweets stored in Redis for the current window of data collection. The logic of IDCAP's support for incremental analytics applies a query on both the tweets in the

current window (via real-time calculations) and the previously collected tweets of an event (via batch processing). After completing a query request, the results of both the real-time analysis and the batch processing are shown in a unified view.

In the Incremental Analytics tab, the first step is to choose an event from a list of all active events; this list is populated by the IDCA App interactively on demand. As shown in Fig. 6.3.1, the 2016 Zika Virus event was chosen and that selection triggered the display of a list of tweet distribution options. Those options are keyword, geo, and all tweets (see Fig. 6.3.2). For illustration purposes, each of these options were respectively selected and their results are shown below.

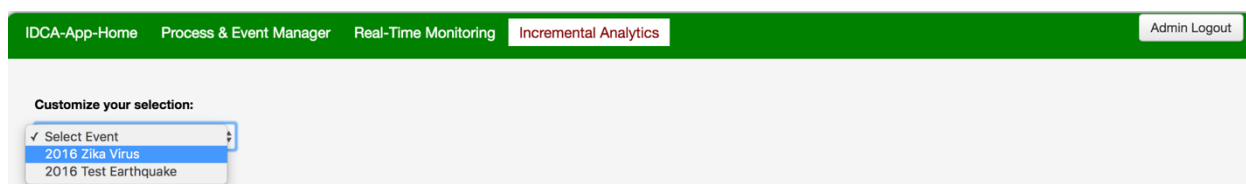


Figure 6.3.1

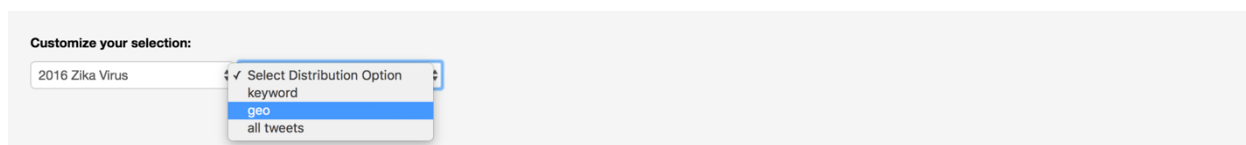


Figure 6.3.2

When the keyword option is selected (see Fig. 6.3.3), the IDCA App interactively displayed a list of event keywords to select. After the #ZikaVirus keyword was selected, the IDCA App performed an incremental analytics query on the 2016 Zika Virus event and provided the number of tweets that contain the #ZikaVirus keyword based on tweet collection days (in this case 17 days).

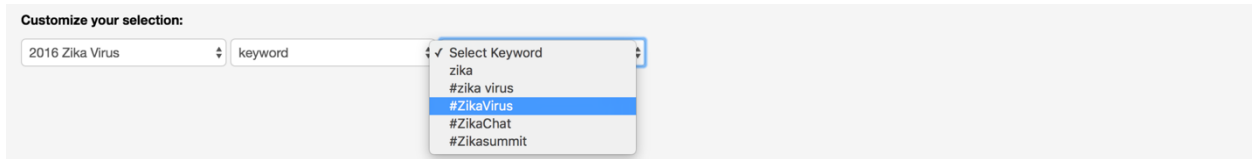


Figure 6.3.3



Figure 6.3.4

As shown in Fig. 6.3.4, both batch and real-time query execution times are provided. The batch results are shown on the left side with blue column bars and its results were calculated by making use of indexes stored in Event_Abstractions Cassandra column family within 0.178 seconds. On the right side, the real-time results are shown with red column bars. For example, 299 tweets were tweeted on the day of 2016127 contain #ZikaVirus keyword. The real-time result are processed in 0.002 seconds by making use of Redis data structures that stores the 2016 Zika Virus event's current collection window. In this case, multiple days are shown in the results because the current window contains tweets from multiple days. Note: the batch processing speed is so fast due to the work I performed in designing the Event_Abstractions column family. Even if this event had millions of tweets, I would still expect similar

performance since the indexes in that column family are designed to answer this type of question directly.

When geo option was selected (see fig. 6.3.5), the IDCA App performed an incremental analytics query on the 2016 Zika Virus event and provided the number of tweets that contain geo-location information over the 17 days of data collection.

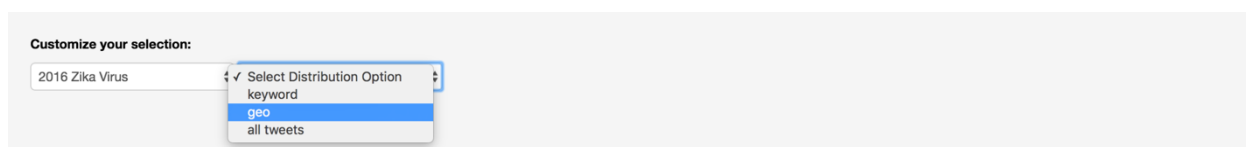


Figure 6.3.5



Figure 6.3.6

As shown in Fig. 6.3.6, the batch results were processed in 0.02 seconds and during the first day of data collection (2016113) 9 geotagged tweets were collected. Also, on the days of 2016119 and 2016124 no geotagged tweets were collected. The real-time results were processed in 0.002 seconds.

When the all tweets option was selected (see fig. 6.3.7), the IDCA App performed an incremental analytics query on the 2016 Zika Virus event (which contained 55,514 tweets at the time this query was performed) and displayed the total tweet count for each day of the data collection.

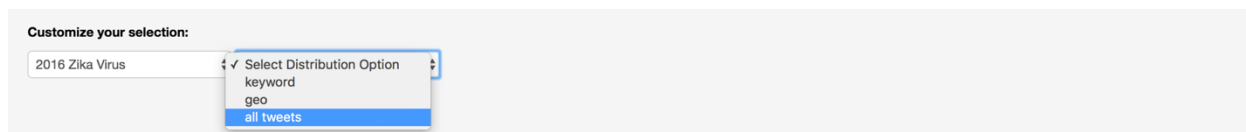


Figure 6.3.7

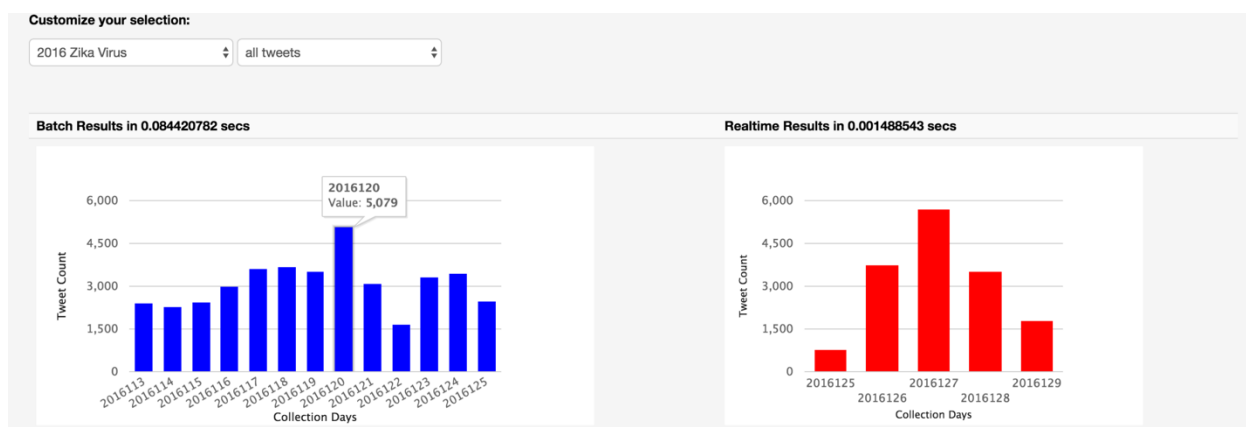


Figure 6.3.8

Figure 6.3: IDCA App Incremental Analytics tab view

As shown in Fig. 6.3.8, the batch results were processed in 0.084 seconds and on the day of 2016120, 5,079 tweets were collected. The real-time results were processed in 0.001 seconds.

6.4 Discussion

The IDCA App is not meant to be a general purpose application for an analyst's day-to-day work. Instead, it is meant to demonstrate that my work on the new data model for EPIC Collect has allowed me to achieve the goals set out for my dissertation by my research questions.

That is, it is possible to answer queries on large data sets at interactive speeds and to apply those queries both via real-time computations on streaming, in-memory data as well as via the batch processing of large data sets stored on disk. The IDCA App was developed in the context of the existence of EPIC Analyze; for instance, there is no need to implement the ability to browse or annotate tweets in the IDCA App since that feature is already present in EPIC Analyze. Instead, now that the new data model for EPIC Collect has been shown to be reliable and efficient (see Chapter 7), it is time to migrate the existing EPIC Analyze software to use that new data model and to integrate the new features of the IDCA App into EPIC Analyze. I now turn to presenting the formal evaluation of my new data model and the IDCAP.

CHAPTER 7

EVALUATION

In this chapter, the results of IDCAP’s evaluation are presented. To evaluate the IDCAP, a variety of evaluation tasks were performed that primarily focused on quantitative aspects of the IDCAP that can be compared with the existing versions of EPIC Collect and EPIC Analyze (and, in particular, Apache Solr which EPIC Analyze relies on for the majority of its functionality).

7.1 Evaluation of the Underlying Cassandra Column Families

To evaluate the effectiveness of IDCAP’s new column family design, we migrated five existing Project EPIC datasets to the new format. The size of each event and the time of the migration is shown in Table 7.1. We selected these events to demonstrate the ability of the new column family design to handle Twitter data sets across three orders of magnitude: hundreds of thousands of tweets, millions of tweets, and tens of millions of tweets. The migration time is, unfortunately, not linear in terms of the size of the dataset. The reason for this is due to the variance in row size that can be found in the old column family design of Filter_Tweet. Some rows in Filter_Tweet contain just one tweet (meaning that only one tweet with a given keyword was generated that day) while others contain tens of thousands of tweets. If one million tweets with a given keyword was generated in a given day, then the use of the tag in our row key design for the Filter_Tweet column family discussed in Chapter 2, would divide those tweets across 16 separate row keys containing roughly 62.5K tweets each. The “wide rows” that contain tens of thousands of tweets take more time to process since you must issue multiple API calls to

Cassandra to retrieve them in batches of 100 tweets each. The increased API calls for these rows also increases the chances for networking errors that then require retries for each failed call. All of this combines to increase the migration time in a way that is nonlinear with respect to the overall size of the data set. However, having migrated these tables, we can now perform an evaluation and comparison of the two column family designs along several dimensions: tweet duplication, tweet distribution by row keys, and the ability to support data analytics via batch processing. Each of these evaluations is presented next.

Table 7.1: EPIC Collect events and indexing times

Event name	Epic Collect tweet count	Indexing time (seconds)
2012 Casa Grande Explosion	183,738	157
2013 Japan Earthquake	1,938,385	2,772
2013 Winter Storm Nemo	4,467,517	9,656
2013 Typhoon Philippines	14,260,178	78,565
2012 Hurricane Sandy	26,792,545	300,865

7.1.1 Tweet Duplication

As discussed above, EPIC Collect stores the same tweet multiple times in the Filter_Tweet column family if a tweet contains multiple keywords. The purpose of that design was to make it easy to retrieve tweets that contained a particular keyword that appeared on a particular date or date range. However, this approach makes it more difficult to compute metrics across an entire dataset since one has to perform extra processing to account for duplicate tweets. On one hand, tweet duplication is not a major concern with respect to storage since disk space is

cheap and plentiful. On the other hand, the time that is spent processing datasets that contain millions of duplicate tweets cannot be underestimated.

As shown in Table 7.2, the amount of disk space consumed by duplicate tweets in large datasets can approach tens of (wasted) gigabytes. The new design of the Event_Tweets column family eliminates tweet duplication within an event as a tweet is guaranteed to only be stored once per event. The amount of disk space consumed by duplicate tweets across events using the new column family design is negligible, since such duplication is rare. In the six years that Project EPIC has been collecting data, we collect an average of 10-20 events at once and since the keywords for each event are specific to the event (place names, event-specific hashtags, etc.) the number of duplicate tweets generated by the new schema will be vanishingly small when compared to the previous approach. Thus, using the new column family, duplicate tweets within an event goes to zero, all space previously wasted is reduced to zero, and when computing dataset-wide metrics, no extra time is spent avoiding duplicate tweets.

Table 7.2: Tweet duplication count and size

Event name	EPIC Collect Tweet Count	Event_Tweets Tweet Count	Tweet Duplication Count / Size in MB	
2012 Casa Grande Explosion	183,738	183,649	89	0.44
2013 Japan Earthquake	1,938,385	1,930,168	8,217	41
2013 Winter Storm Nemo	4,467,517	4,252,790	214,727	1,074
2013 Typhoon Philippines	14,260,178	11,876,836	2,383,342	11,916
2012 Hurricane Sandy	26,792,545	22,150,275	4,642,270	23,211

7.1.2 Tweet Distribution by Row keys

As discussed in Chapter 2, EPIC Collect stores tweets in the Filter_Tweet column family that contain a given keyword and were collected on a given day across sixteen rows using the `keyword:juliandate:tag` format. It does this to ensure an even distribution of tweets across a cluster of machines. Without the tag, there is a danger of every tweet collected on a given day with a given keyword being stored on just a single node in the cluster and replicated to just one additional node in that cluster. With the tag, sixteen rows are generated with tweets being evenly distributed across them; those rows are then evenly distributed across the nodes of the cluster. This row key design is thus an attempt to allow Cassandra to keep its entire cluster of machines fully utilized rather than directing all work to just a few nodes [Anderson et al. 2015]. While that was the intent, Table 7.3 shows that the nature of Filter_Tweet design with respect to the use of wide rows leads to an uneven distribution of tweets across row keys. Many rows in a data set have just a few tweets and are thus causing Cassandra to do too much work to process the tweets after they are stored; this work comes from the fact that sixteen API requests may be needed to retrieve 16 tweets (in the case where each row stores just a single tweet) when all of them could have been stored together and retrieved in just a single API call. On the other side of the distribution, a few rows are being created with hundreds of thousands of tweets requiring many API calls to process and going way over the recommended maximum size of 100MB for a given row. Anytime these rows need to be processed to answer a query, performance suffers; indeed, it is these wide rows that led to the nonlinear increase in indexing times that we discussed.

Table 7.3: Tweet Distribution by row keys

	2012 Casa Grande Explosion	2013 Japan Earthquake	2013 Winter Storm Nemo	2013 Typhoon Philippines	2012 Hurricane Sandy
Tweet Count	Number of Filter_Tweet column family row keys that contains tweet count				
1-10	66	7,270	8,493	11,809	6,106
11-100	160	322	1,696	12,499	6,171
101-1000	103	3,471	1,852	5,442	3,701
1001-5000	89	193	1,167	3,871	3,432
5001-10000	-	-	32	167	281
10001-20000	-	-	65	115	123
20001-50000	-	-	31	62	109
50001-100000	-	-	-	-	112
100001-150000	-	-	-	-	16
150001-173000	-	-	-	-	16
total rowkey count	418	11,256	13,336	33,965	20,067

7.1.3 Row key Generation

Table 7.4 demonstrates another advantage to the new column family design. With the design of the Filter_Tweet column family, there is no way to know how many rows are associated with a given data collection event. If you need to determine, for instance, how many tweets have been collected for a given dataset, you must first generate all possible row keys for that event and then ask Cassandra which of these possible rows actually exist and then count the tweets that are in that row by iterating over all of its columns. The data in Table 7.4 shows that

this process takes a non-trivial amount of time for large data sets. For example, the row key generation process takes more than seven minutes for our Hurricane Sandy dataset with ~22M tweets.

Table 7.4: EPIC Collect row key generation time

Event name	Filter_Tweet		Event_Tweets	
	Row key count	Row key generation time (seconds)	Row key count	Row key generation time
2012 Casa Grande Explosion	418	5.9	10	Constant
2013 Japan Earthquake	11,256	103	97	Constant
2013 Winter Storm Nemo	13,336	175	213	Constant
2013 Typhoon Philippines	33,965	233	594	Constant
2012 Hurricane Sandy	20,067	421	1,108	Constant

The other aspect of the new design that becomes clear from the data in Table 7.4 is that the new design results in significantly less rows per dataset. Rather than having a haphazard distribution of tweets by keywords and collection days (as shown in Table 7.3 for Filter_Tweet), each row of an event in the Event_Tweets column family has 20K tweets in it, thus the 2012 Casa Grande Explosion event with ~183K tweets can be stored in just 10 rows compared to the 418 rows that were generated for the Filter_Tweet column family. Furthermore, each row key generated for an event is stored in the Event_Information column family in the et_rowkeys column as shown above in Figure 5.3. Using the new column family design, determining the number of row keys for a data collection is a constant time operation requiring one API call to retrieve the value of the et_rowkeys column for a given event and then determining the length

of the array that is returned. Therefore, the new column family design leads to better tweet distribution, significantly fewer row keys overall, and the elimination of the row key generation step required by the previous design.

7.2 Batch Data Processing

The second aspect of our evaluation is to compare how the two column family designs perform when supporting data analytics via batch processing. In this evaluation, we carefully prepared a set of queries that can be executed using batch processing for both of the column family designs, the original model used by EPIC Collect and the new model presented in this thesis. Given the structure of these column families, it is possible to describe the steps that are required to compute each of the queries. We do this next and then move on to discussing the results of our evaluation.

The original EPIC Collect column families are `Event_Filter` and `Filter_Tweet`; for these column families, the following steps are performed for all data analysis tasks that take place via batch processing: (1) generate all possible `Filter_Tweet` row keys that might contain tweets for a particular event based on the keyword and date information stored in `Event_Filter` column family; (2) check if each generated row key exists in the `Filter_Tweet` column family; (3) for each existing row key, retrieve all of its associated columns; (4) perform the required analysis task on the tweet objects that are stored in those columns; (5) store the results of the analysis in a global data structure that keeps track of accumulated results as the iteration over row keys and columns is performed.

The new column families are `Event_Tweets`, `Event_Abstractions`, and `Event_Information`; for these column families, the following steps are performed for all data analysis tasks that take place via batch processing: (1) retrieve the array that contains all row

keys generated for the event of interest from the Event_Information column family; (2) based on the query, access the relevant index in the Event_Abstractions column family—
`event_name:jd_keywords`, `event_name:jd_geotagged`, and `event_name:jd_index`—and use elements of the query to access the relevant tweet references in that index; (3) at this point, many queries that do not require tweet metadata can be readily answered (such as “how many tweets contain the keyword “tornado”) and processing can stop since the answer can be computed directly from the index; otherwise, take the tweet references returned in step 2 and use them to retrieve the tweet metadata of each reference which is stored in the Event_Tweets column family; (4) perform the required data analysis on the returned tweets and return the result; there is no need for a global data structure that stores accumulated results since the index has taken us directly to the tweets that contain the answer we seek.

While the total number of steps differs by just one between the two column family designs, the difference in execution time is significant. Our evaluation made use of the nine queries shown in Table 7.5. The first eight queries perform a variety of common counting and filtering tasks. The last query is one which asks the column family design to deliver ALL tweets related to event so that some exploratory data analysis task can then be performed on those tweets, such as searching, sorting, clustering, sampling, and so on. Each query was performed on the two column family designs using the steps enumerated above. The time used to perform a query is recorded; each query was performed ten times using the same set of test machines and the average time across all ten runs are provided in the results below. Each query was applied to the five Project EPIC datasets listed in Table 7.2. This means that nine queries were applied ten times each to five different datasets stored using two different column family designs; our evaluation thus consisted of executing a total of 900 queries to generate our results.

Table 7.5: The Queries used in the Evaluation

Q1	Count unique tweets
Q2	Count unique geo-tagged tweets
Q3	Count unique tweets that were tweeted in a specified date range
Q4	Count unique tweets that contain a specified event collection keyword
Q5	Retrieve unique geo-tagged tweets
Q6	Retrieve tweets that were created in a specified date range
Q7	Retrieve tweets that contain a specified event collection keyword
Q8	Retrieve tweets that were created in a specified date range and contain a specified keyword
Q9	Retrieve all unique tweets of a specified data set for exploratory analytics

In Table 7.6, the results of the evaluation queries are shown when applied to the 2012 Casa Grande Explosion dataset. This dataset was collected in twelve days starting from 2012362 to 2013007. The table shows how many seconds it takes to perform a particular query in seconds for the original and new column family designs, the number of tweets that were involved with that query (rounded to the nearest thousand) and the number of times faster the new column family design is over the original column family design. While each of our queries remains the same across each dataset, the constants associated with the queries may be different across datasets; typically, the date range corresponds to the most active portion of the event while the selected keyword was the most commonly used keyword for the event. Thus, for this dataset Q3, Q6, and Q8 used as the date range the first four days of the event. In Q4 and Q7, the keyword that was used was “Arizona.” Finally, Q9, shows the time it takes to retrieve all unique tweets of the dataset. These tweets can then be used to perform some other data analysis task. Based on the

results, the new column families provide fast batch data processing with respect to counting cardinality of tweets at least 97 at most 891 times, with respect to filtering at least 11 at most 14 times, and in exploratory analytics 10 times faster than EPIC Collect column families. It is clear from the results that the new column family design is significantly faster than the original column family. Counting-related tasks are essentially performed in constant time and can be up to 891 times faster than the same operation performed on the original column family design and up to 500 times faster on average. With respect to tweet retrieval (either retrieving all tweets or performing filtering and then retrieval), the new column family design is, on average, twelve times faster than the original column family for this dataset.

Table 7.6: 2012 Casa Grande Explosion query results

2012 Casa Grande Explosion	Counting				Filtering				Exploratory
	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
Original Column Families (seconds)	312	126	71	33	174	130	354	126	427
New Column Families (seconds)	0.35	0.16	0.32	0.34	12.4	11.7	30	10.5	39.85
Query Result Tweet count	183K	56K	66K	179K	56K	66K	179K	65K	183K
Times Faster	891.4	787.5	221.9	97	14	11.1	11.8	12	10.7

In Table 7.7, the evaluation results for the 2013 Japan Earthquake dataset are presented. This dataset contains 1.9M tweets collected across 228 days starting from 2013298 to 2014160. The date range used in Q3, Q6, and Q8 was the first five days of the event and the selected keyword for Q4 and Q7 was “tsunami.” With this dataset, it was no longer possible for the new

design to perform counting in constant time, the arrays of tweet ids were significantly larger and so costs were incurred in pulling those arrays out of Cassandra and into the main memory of the program performing the query. Nevertheless, the new design was on average 882 times faster at counting than the original design and nearly 2000 times faster in the case of Q2. With respect to tweet filtering and retrieval, the new column family design was on average 9.2 times faster than the original design.

Table 7.7: 2013 Japan Earthquake query results

2013 Japan Earthquake	Counting				Filtering				Exploratory
	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
Original Column Families (seconds)	3242	1345	499	3.4	1748	923	28	29.4	4,487
New Column Families(seconds)	2.4	0.69	2.17	2.3	112	93	5.21	5.2	438
Query Result Tweet count	1.9M	446K	465K	15K	446K	465K	15K	15K	1,930,168
Times Faster	1,350	1,949	229	1.4	15	9.9	5.3	5.6	10.24

In Table 7.8, the results of the evaluation for the “2013 Winter Storm Nemo” dataset are shown. This dataset contains 4.25M tweets and was collected across 58 days from 2013038 to 2013095. The date range used in Q3 was the last 16 days of the event; whereas in Q6 and Q8, the date range used was the first 15 days of the event. The reason for this was to capture the most active collection times for this event. The keyword used in Q4 and Q7 was “winter storm nemo.” Nevertheless, the new design was on average 1033 times faster at counting than the original

design. With respect to tweet filtering and retrieval, the new column family design was on average 10.6 times faster than the original design.

Table 7.8: 2013 Winter Storm Nemo query results

2013 Winter Storm Nemo	Counting				Filtering				Exploratory
	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
Original Column Families (seconds)	7086	3432	4963	6.2	4689	3467	57	50	9747
New Column Families (seconds)	5.1	2.2	4.2	5.3	285	246	11	10	805
Query Result Tweet count	4.2M	1.4M	1.1M	26K	1.4M	1.6M	26K	24K	4,252,790
Times Faster	1389	1560	1181	1.1	16.45	14.09	5.18	5	12.10

In Table 7.9, the results for the “2013 Typhoon Philippines” dataset are shown and the results for the “2012 Hurricane Sandy” dataset are shown in Table 7.10. Despite these events being significantly larger than the previous datasets (11.8M and 22.1M respectively), the speed-ups of the new design over the original design remain approximately the same: ~1000 times faster with respect to counting and ~10 times faster with respect to filtering and retrieval. These results will translate into a “night and day” difference when using EPIC Analyze to work with datasets stored using the new column family design. These differences turn queries that can take 7 hours to execute on the original column family design and transform them into queries that return a result in 14 seconds (as seen in Table 7.10 for Q1). It is the difference between a data analysis session that seems laborious and one that is interactive (in comparison). It is a significant achievement that will positively impact the lives of Project EPIC analysts.

These results are even more significant, in that these times do not include the row-key generation time that was discussed above for the original table design. We chose to exclude that time from the results to allow for a closer “apples to apples” comparison between the two designs. If we add in this time, the differences between the two designs are even more significant.

Table 7.9: 2013 Typhoon Philippines query results

2013 Typhoon Philippines	Counting				Filtering				Explorator y
	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
Original Column Families (seconds)	25619	12697	3759	247	17005	5815	2093	15.67	31175
New Column Families (seconds)	14.82	6.3	12.9	14.34	1016	441	242	14.3	2369
Query Result Tweet count	11M	4.7M	2.4M	1M	4.7M	2.4M	1M	7K	11,876,836
Times Faster	1728	2015	291	17.22	16.73	13.18	8.6	1.09	13.15

Table 7.10: 2012 Hurricane Sandy query results

2012 Hurricane Sandy	Counting				Filtering				Exploratory
	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
Original Column Families (seconds)	58123	25144	9300	3310	31529	15305	38012	10339	57779
New Column Families (seconds)	32	11.2	27	38	1587	1048	2889	709	4278
Query Result Tweet count	22.1M	7M	5.7M	15M	7M	5.7M	15.3M	3.7M	22,150,275
Times Faster	1816	2245	344	87	19.8	14.6	13.15	14.58	13.5

The price for this significant speed-up is, of course, the time it takes to index the tweets when storing a dataset in the new column family format. We are able to provide significantly faster batch processing times over the original design because a lot of the work is done up front rather than per query. To show this, in Table 7.11, we show how long it takes to index each of our five datasets for the new column family design and compare that with the time it takes to retrieve all tweets for a data set using the old column family design. The former shows the upfront costs that must be paid to have efficient queries while the latter shows the costs (per query) for not performing the work associated with creating indexes. As can be seen in Table 7.11, the total time to create the indexes for data sets consisting of millions of tweets is less (in some cases significantly less) than the time it takes to perform a retrieval of an entire data set. For datasets in the tens of millions of tweets, the indexing time is significantly higher but this cost only needs to be paid once. Fortunately, with both of these datasets, an analyst has only to

perform a handful of analysis tasks that require the entire data set (three such queries for 2013 Typhoon Philippines and six queries for 2012 Hurricane Sandy) for this indexing cost to be repaid.

Table 7.11: EPIC Collect Q9 vs Indexing in seconds

Event name	New Column Family Indexing Times (seconds)	Original Column Family Q9 processing times (seconds)
2012 Casa Grande Explosion (183K)	157	427
2013 Japan Earthquake (1.9M)	2,772	4,487
2013 Winter Storm Nemo (4.2M)	9,656	9,747
2013 Typhoon Philippines (11.8M)	78,565	31,175
2012 Hurricane Sandy (22.1M)	300,865	57,779

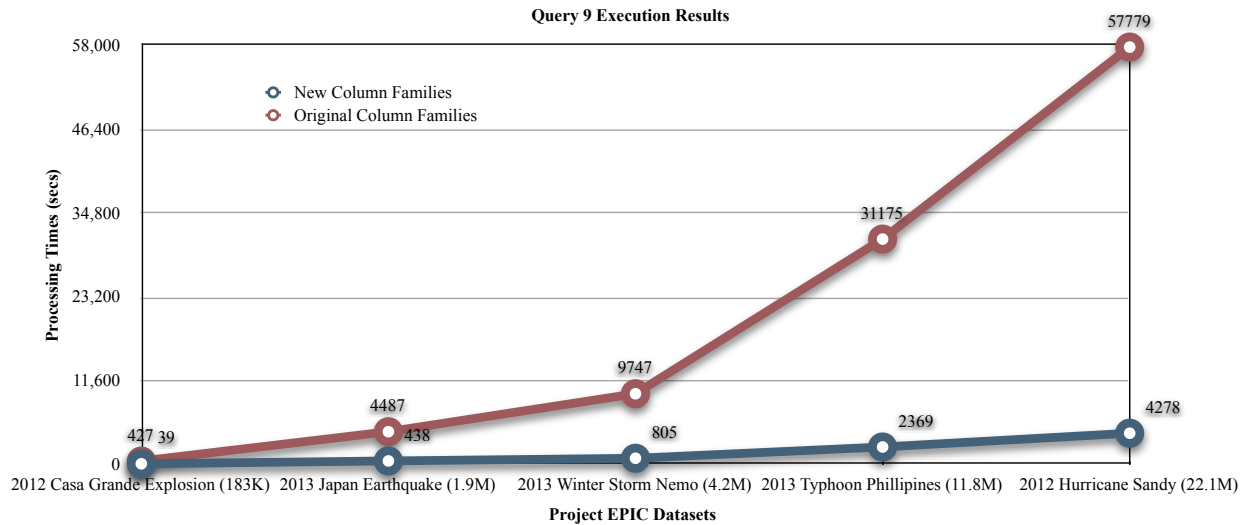


Figure 7.1: Query 9 Execution Results in seconds

As a final means of comparison, Figure 7.1 shows graphically the significant difference between the execution times in performing Q9 for the original and new column family designs. The new design avoids the exponential growth that was present in the original design with respect to performing queries that need to process the entire dataset. These results demonstrate the benefits that can arise with careful data modeling in the context of data-intensive software systems.

7.3 Application (System) Level Batch Processing

The final aspect of our batch-oriented data processing evaluation is to compare the exploratory batch processing capability of the IDCAP with EPIC Analyze (Apache Solr) and EPIC Collect. The IDCAP and EPIC Collect provide their own indexing mechanism, and their batch data processing steps are enumerated above. On the other hand, EPIC Analyze makes use of Apache Solr which creates a Lucene full-text index for these datasets. My evaluation will allow me to compare my work with Apache Solr, a well known and widely used framework.

Apache Solr can be used to perform batch processing via the following steps: (1) generate/update a Solr query that is constructed using parameters such as the event name of interest, tweet id, text, user screen name, as well as the number of documents that should be returned, and Solr's CursorMark parameter; (2) submit this query to Solr; (3) check if all possible Solr documents (i.e. matching tweet objects) were received using nextCursorMark parameter; (4) perform the required analysis task on the tweet objects that were received; (5) store the query results in a global data structure that keeps track of accumulated results as the iteration over solr documents is performed and (6) perform all steps between 1 and 5 until all documents are received and processed. This procedure is necessary because a Solr query can match millions of documents and it is inefficient to return all of them at once; instead, Solr

returns just a subset of documents with each query (10 by default) and the CursorMark parameter is used to keep track of which results have been seen for a given query.

To evaluate the three systems, three Project EPIC datasets (see Table 7.12) were selected; each dataset was indexed in the three systems with their own indexing mechanism. To evaluate these systems, the following query (Q10) was used: “calculate the user tweet count distribution of all tweets for a particular event”. That is, for a given event, find all unique users in the event and then calculate how many tweets each user contributed to the event. These distributions follow a power law since there are a few users for any event that generate a huge number of tweets and then many users that generate just a single tweet. This query was selected because it can not be directly answered by any of these systems using their indexes. Therefore, to answer this question, each system must perform more work than simply looking up answers in pre-computed indexes.

This query was performed on each system by using the enumerated steps above. The time spent to perform the query was recorded. Each query was performed five times using the same set of test machines and the average time across five runs are provided in Table 7.12.

Table 7.12: Query 10 Execution Results in seconds

Event Name	IDCAP	EPIC Collect	Apache Solr
2012 Casa Grande Explosion	106	386	1,553
2013 Winter Storm Nemo	3,934	21,966	41,492
2013 Typhoon Philippines	10,975	32,166	196,061

As mentioned above, EPIC Analyze makes use of Apache Solr to answer analyst queries at interactive speeds since EPIC Analyze only ever displays 50 query results at any one time due

to a carefully-designed pagination mechanism. Therefore, retrieving small chunks of query results by Apache Solr does not compromise analysis tasks. However, when it comes time to read all of the tweets of one of Project EPIC’s large datasets to create a hash map of users that tracks their contribution to that dataset, the IDCAP performs significantly faster than EPIC Collect and Apache Solr (see Fig 7.2). According to the results shown in that figure, the IDCAP is at least 2.93 times and at most 5.58 times faster than EPIC Collect; and at least 10.54 times and at most 17.86 times faster than Apache Solr.

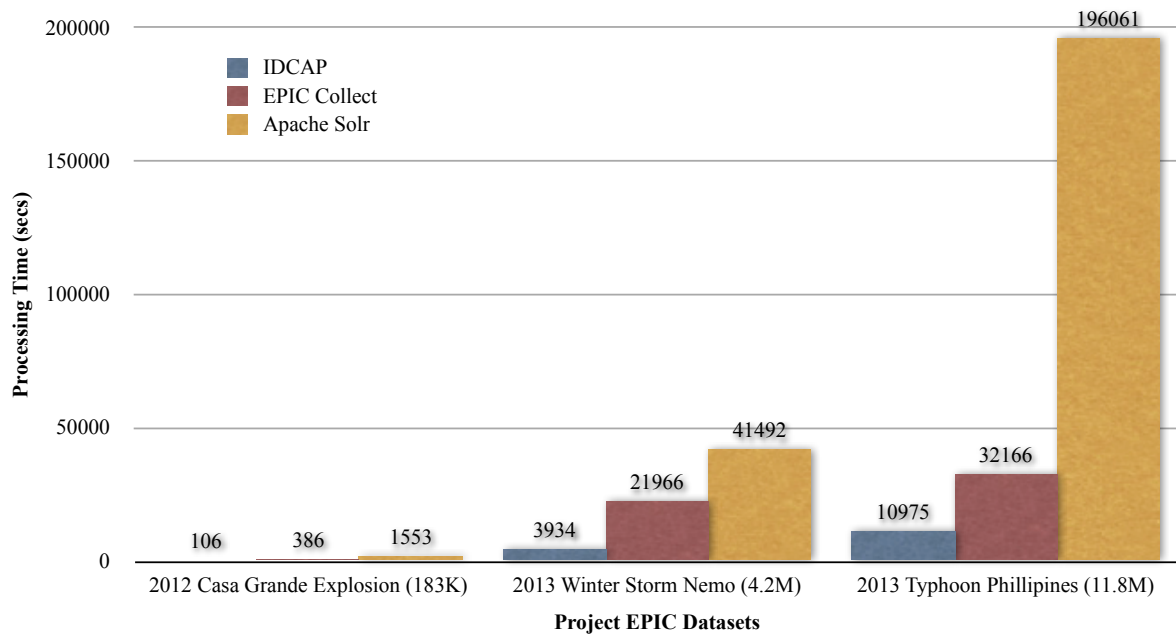


Figure 7.2: Query 10 Execution Results in seconds

7.4 Discussion

The results of IDCAP’s evaluation demonstrate the significant benefits of carefully designing the data model for an application domain. The new column family design significantly outperforms what is achievable when working with the existing EPIC Collect and EPIC Analyze systems. My new data model was built with full knowledge of the existing limitations of the

original column family design. While those systems provided significant benefits of their own, their limitations would show with the long running times of batch processing queries and, furthermore, real-time analytics on tweets streaming in for an active data collection event *was not even possible*.

Of course, my new data model is not a panacea; it was designed with a particular application domain and with full knowledge of the types of queries that are common for that domain. While the IDCAP can compute the number of geolocated tweets contained in a large Twitter dataset in less than a second, it cannot compute the answer to a full-text search query. However, a rudimentary full-text search could be built on top of the new data model by first using other attributes to narrow a result set from millions of tweets to hundreds and then using string-based search methods on the remaining tweets. Although, a better approach would be to just use Apache Solr to handle full-text search as is already implemented in EPIC Analyze. There is little point in trying to design a data model that attempts to optimize for all possible queries, since that is impossible.

However, it is important to note that the techniques used to generate the new data model are generalizable. The avoidance of duplicate data; the use of one column family to serve as an index for a second column family; the self-imposed limits on row size; and the use of domain-informed row keys are all techniques that can be used to generate column family designs for other application domains. Furthermore, the design of the Event_Abstractions column family is generic enough that many different indexes can be built and stored within it. In previous work [Aydin and Anderson 2015], I designed an approach to incrementally sort large Twitter data sets via batch processing such that millions of tweets could then be displayed in sorted order along a number of sort dimensions within EPIC Analyze at interactive speeds. The indexes created by

ISM could easily be added to the Event_Abststractions column family as could an index that pre-computes the answer to Q10 above for any data set collected by the IDCAP. Therefore, I believe my data modeling techniques are generic enough to be of use by other designers of data intensive software systems.

CHAPTER 8

RELATED WORK

In this chapter, related work is presented under two categories: data-intensive system design, and data modeling for NoSQL databases.

8.1 Data Intensive Systems Design

This thesis work builds on the work that has been invested in the original work to design, develop, and deploy EPIC Collect [Anderson and Schram 2011; Schram and Anderson 2012; Anderson et al. 2013] and EPIC Analyze [Anderson et al. 2015] as discussed above. This work was a significant achievement at the time, providing new insights into the software architectures required to make data-intensive software systems reliable and scalable and tackling the initial thorny work that was required to identify a data model that supports those characteristics for the reliable collection of large Twitter datasets. With the benefit of hindsight, our work developing the new data model addresses the issues that prevented the original data model from directly supporting efficient data analysis. The sections above have already outlined our contributions in this regard (elimination of tweet duplication, elimination of row key generation, elimination of inefficient wide rows that went beyond recommended limits and took too many API calls to process efficiently, the addition of a “big picture” view of collected data sets, the support for efficient data analysis for the most commonly requested metrics, the ability to incrementally add

missing tweets at any time, and the automatic ordering of tweets by creation date). We now discuss additional related work.

In [McTaggart 2012], McTaggart designed and implemented a web application for Twitter analytics and then conducted a usability study to elicit the questions that social media analysts ask with respect to Twitter data collected for disaster events. McTaggart mentions that the single largest issue with her prototype implementation is performance at scale. Preliminary numbers from her working prototype showed that queries performed against forty thousand tweets could take anywhere from 10 seconds to several minutes. This slow performance was problematic when computing queries on datasets consisting of 20 million tweets. In McTaggart's research, Project EPIC analysts were interviewed to identify different research interests, figure out each analyst's requirements and desires for an analysis tool, and then generate a general set of requirements for a data analytics platform in support of crisis informatics research. Moreover, McTaggart provides different types of techniques that are used in data analytics. In thesis work, her findings are very useful to understand the needs of Project EPIC analysts. Indeed, the big picture view provided by the Event_Information column family and the queries that are supported by the Event_Abstractions table were all influenced by McTaggart's findings.

In [Anderson et al. 2015], design, development and data modeling details of EPIC Analyze is provided. EPIC Analyze—Project EPIC's batch data analytics software—is a flexible data analysis environment accessed via a web application for viewing, searching, filtering, annotating, and visualizing Project EPIC's Twitter data sets at interactive speeds. EPIC Analyze is designed to address issues indicated in [McTaggart 2012] and identified open source frameworks such as Hadoop, Cassandra, Solr, Pig, Redis, and PostgreSQL that could be used together to meet scalability and performance goals for a general purpose data analysis

environment. EPIC Analyze indexes the datasets stored in EPIC Collect in Apache Solr and then relies on Solr to perform full-text search and filtering operations. Our goal in designing the new data model for EPIC Collect was to help reduce EPIC Analyze's reliance on Solr for all of its data analysis needs [Barrenechea et al. 2015]. Now, the new support for data analysis provided by the `Event_Information` and `Event_Abstractions` column families can be used directly to answer a wide range of queries more efficiently than Solr and now Solr is used just for its primary purpose of providing efficient full-text search.

In [Andreolini et al. 2011], a software architecture for the analysis of cloud-based data streams is proposed. The proposed architecture's aim is to support system management of large enterprise data centers for cloud based infrastructures and to analyze the data collected to glean useful information about the state of the system in an efficient manner. The technologies used in the architecture aim to achieve scalability by increasing the number of software and hardware components and the reliability of the processes for collection and analysis. Our work shares the same goal of storing, processing, and analyzing large amounts of information but the type of information is different. In [Andreolini et al. 2011], the work focuses on system monitoring---resource utilization, system response times, and throughput---by keeping system related information stored in HDFS and HBase clusters and analyzing them using MapReduce jobs generated by programs written using Apache Pig. We focus instead on crisis data sets consisting of large amounts of Twitter data. We do, however, make use of similar techniques; for instance, our work on ISM [Aydin and Anderson 2015] sorts tweets using MapReduce jobs generated by the DataStax Enterprise version of Apache Pig which can read/write data stored in Cassandra.

In [Cameron et al. 2012] an Emergency Situation Awareness-Automated Web Text Mining (ESA-AWTM) platform was presented and it is designed for crisis coordinators in the

Australian Government's Crisis Coordination Centre (CCC). The goal of this system is to identify situational awareness information from tweets generated during the response phase of crisis events. The ESA-AWTM platform's interface allows users to monitor and refresh alerts related to queries of interest. The Burst Detector/Alert Monitor interface provides stylized words in order to visualize incident status based on statistical models. The Cluster visualizer summarizes situational awareness information from streamed tweets (tweeted in Australia and New Zealand) by using the Carrot Clustering engine and Solr for watch officers. Support Vector Machines are trained to detect high-value messages such as "infrastructure damage". We share the same goals of designing systems for emergencies. In our system, we would like to provide these types of queries provided by ESA-AWTM in our future work.

In [Oussalah et al. 2013], a software architecture for collecting and analyzing geospatial and semantic information from Twitter data was described. The tweets were consumed by a Twitter4j Java application and then transferred into a PostgreSQL database using the PostGIS spatial extension. Wordnet and Solr were used to relate tweets together if those tweets share the same meaning. The tweets are then made available by a Django web application, using GeoDjango for geospatial queries and the Haystack API for semantic queries. The goal of this infrastructure was to search for tweets via semantic keywords and coordinates, and export the results via a map or CSV file. The focus of [Oussalah et al. 2013] is on a particular domain-independent analysis technique and not with supporting the entire analysis life cycle for crisis informatics research. In contrast to their work, we store the entire JSON object of collected tweets in a scalable and incremental fashion in Cassandra in a way that allows us to answer all deep queries related to the entire tweet object. Furthermore, the type of analysis performed in [Oussalah et al. 2013] is one that could be incorporated into our environment in a straightforward

manner; Project EPIC’s NLP-related work has focused instead on the creation of machine learning classifiers to identify tweets that contain situation awareness information [Verma et al. 2011].

For a number of years, Purohit et al. has been maintaining Twitris [Purohit and Sheth 2013], a citizen sensing platform for collecting and applying NLP techniques to millions of tweets to glean important information about a variety of events, from entertainment to disaster events. Unfortunately, the developers of Twitris do not reveal the technologies and systems they use to achieve their analysis at this level of scale. They do serve as an example of the types of analysis that we could apply to Project EPIC data sets in the future.

8.2 Data Modeling for NoSQL Databases

In [Aydin and Anderson 2015], I, along with my advisor, explored how support for incremental sorting could be added to EPIC Analyze. One challenge with data-intensive software systems is that “easy” operations (such as sorting) become hard at scale [Anderson 2015; Aydin and Alaghband 2013]. It is difficult to provide an environment that allows a user to browse data sets consisting of millions of tweets; it is even more difficult to then allow those data sets to be sorted on demand. Instead, our approach was to batch sort a large data set along multiple dimensions and then store the order of tweets in a separate column family that could then be used to display the tweets in ascending and descending orders along a wide range of sort dimensions (by Tweet id, screen name, hashtag, etc.) [Aydin and Anderson 2015]. In addition, our sorting method was designed to support the sorting of data sets that are under active collection; it could sort an initial data set all at once and then incrementally sort new tweets into the various sort orders without having to process any of the previously sorted tweets again.

Hence, our approach was called the incremental sorting method (ISM). ISM first uses Hadoop to sort tweets in parallel along a number of dimensions using scripts that were written in Apache Pig. Once the tweets have been sorted by Hadoop, ISM generates an index that can then respond to queries, filtering requests, sort requests, etc. on the data set and respond in near constant time. This index makes use of a similar technique described above for the Event_Abstractions column family. Indeed, moving forward, we will be able to add the indexes created by ISM as additional rows into the Event_Abstractions column family that will allow our new data model to also support sorting alongside its new support for data analysis. In this case, however, these rows cannot be added as tweets are streaming in. Instead, the incremental sorting method currently assumes that it will operate on at least one day's worth of tweets before sorting begins. It would then sort any newly arrived tweets on each subsequent day. This time boundary allows ISM to keep track of "where it left off" so that when it performs a sorting operation it knows which tweets have been sorted and which need to be sorted. The time interval of a day was selected to ensure that there was plenty of time for Hadoop to batch sort large data sets along multiple sort dimensions. This interval can be reduced if EPIC Collect, EPIC Analyze, and the software that implements ISM are deployed on a larger, more powerful cluster than can perform all the sort operations in a shorter period of time. In this way, our work on the new data model nicely complements our previous work on incremental sorting of large data sets. The Event_Abstractions column family serves as a nice home for any functionality that wants to build an index on the tweets that are efficiently stored by the Event_Tweets column family.

In [Jia et al. 2015], a search-efficient hybrid storage system (LuBase) for large-scale text data analytics is proposed. The system integrates Lucene into HBase to take advantage of Lucene's full-text search capability and HBase's high scalability and reliability. Lucene index is

created on immutable part of data and updated parts are stored in HBase and row key component provides mapping. The pre-built full text index allows to provide fast interactive queries on text data. Although we share the goal of providing fast queries, unlike them we don't mainly provide full-text search with our underlying data model since that's provided by EPIC Analyze(Solr).

In [Li et al. 2014], a query-oriented data modeling (QODM) approach for NoSQL databases is proposed that provides a data model and a data schema for an application based on the following inputs: the data query requirements of the application and the stored structure of its data. Furthermore, the authors also provide a platform independent meta-model of a data schema for NoSQL databases. Our work shares common goals with the QODM approach such as considering data modeling not only to store data in a scalable fashion but also to provide fast and efficient query processing, to improve the querying ability of an existing data model by reducing the number of database visits, and to design a data model to reduce data redundancy. On the other hand, unlike the QODM approach, we deliberately designed a new data model to eliminate the issues of an existing one; however, the QODM approach cannot fix issues of an existing data model and those issues may be inherited in a data model generated based on it. As the authors mentioned, a query-oriented data model must reduce the number of visits to the database when querying to increase query performance but they did not provide any results in their evaluation for the querying times of a QODM-generated data model over other approaches. Furthermore, the QODM approach is built on the aggregation of data and indexes to reduce data redundancy in support of fast querying but they did not mention how much data is aggregated and how much space is gained in a QODM generated data schema. Moreover, a data model that works well in one NoSQL database may not perform well with another. For instance, Cassandra is the best match for our needs but HBase, Solr/Lucene, or MongoDB do not meet our requirements

[Anderson 2015] even though we would have the same application-related requirements and data model. There are, thus, many issues that must be reconciled to find the data model and NoSQL database that together provide the best match for a particular application's needs.

In [Mior 2014], a cost-driven automated schema design approach for NoSQL databases is proposed. Although the author's ultimate goal is to develop a tool to automate the process of designing database schema, we share common goals such as optimizing query performance, minimizing storage space, and decreasing the number of steps while answering queries. In this thesis work, we deliberately designed a new data model by considering Project EPIC analysts needs and evaluated our design based on querying time, storage space, and tweet distribution by row key. In our work, we addressed some of the same issues such as handling the side-effect of wide rows, minimizing storage space, and decreasing the number of steps while answering queries. In [Mior 2014], the author mentions that "it is impractical to maintain all views for all possible queries" in term of storage. While that is true, it is possible to design column families that can directly answer all of the most common queries in a particular application domain as we did in our work. Our Event_Abstractions table was carefully designed to allow support for all queries related to a data collection's keywords, a tweet's collection date, and all geotagged tweets but, does not, for example, support full-text search on the content of our tweets. For that, we make use of Solr, a tool designed exactly for that task.

In [Gao and Qiu 2014], a general and customizable indexing framework (IndexedHBase) over distributed NoSQL database (HBase) is presented to achieve efficient queries. We share the same goals of efficiently extracting a requested subset of a larger data set, limiting pre-computations before actual analysis, and providing fast queries on historical and streaming datasets. We both collect, store, and analyze Twitter data sets and create indexes on them but

instead of using HBase we make use of Cassandra. Our Event_Abstractions column family provides three indexes to enable efficient extractions of subsets of datasets such as retrieving geotagged tweets without touching the rest of the data set and by gathering metrics in constant time such as tweet count per day, tweet count by keywords, and geo-tagged tweets per day. We also support range scans based by tweet collection date in parallel. On the other hand, in contrast to their work, we do not provide the ability to create customizable indexing, instead in our approach developers must design and add new rows to the Event_Abstractions column family to provide new indexes that can then be used to support queries made in EPIC Analyze.

CHAPTER 9

FUTURE WORK

Moving forward, our work will focus on adding features to the IDCA App that continue to demonstrate the benefits of the new data model. In particular, we would like to add additional features that allow analysts to issue a wider range of customizable queries including the ability to have a single query access more than one index stored in the Event_Abstractions column family. In addition, we plan to add the capability to view the tweets that match the query (as opposed to just seeing summary statistics) and to specify a time range for the query. In addition, we will add the ability to export the results of a query to an external data format such as CSV or JSON. Indeed, many of these features are already present in EPIC Analyze, so a more important piece of future work will involve migrating that system to make use of the new data model while still making use of Apache Solr for full-text search and PostgreSQL for storing annotations on tweets.

Furthermore, we would like to develop algorithms that incrementally process streaming tweets since the IDCAP provides a feasible infrastructure to perform such algorithms. For example, we are interested in developing a real-time version of the incremental sorting method (which currently relies on batch processing techniques to sort large amounts of data) and to explore the creation of real-time visualization of metrics important to crisis informatics researchers.

CHAPTER 10

CONCLUSIONS

In this thesis, a novel and well-designed software infrastructure called the Incremental Data Collection and Analytics Platform (IDCAP) was designed. We have shown how various data analytics frameworks such as Cassandra, Spark, Redis, and RabbitMQ can be efficiently deployed together in support of real-time data collection and analytics. The IDCAP shows that heterogeneity is inevitable in such data collection and analytics platforms.

The IDCAP can index streaming data in real-time and perform real-time data analytics on the current window at interactive speeds. Its underlying data model successfully addressed the limitations of the original data model for EPIC Collect, eliminated tweet duplication, and supports a wide range of common queries in an efficient manner. This data model is composed of three new Cassandra column families—Event_Tweets, Event_Information, and Event_Abstractions.

The Event_Tweets column family stores tweets compactly and without duplication within an event in a scalable fashion. In the original data model, tweet ids were stored as strings; in the new data model, tweet ids are stored as 64-bit integers taking much less space and automatically sorting tweets in terms of creation date. The rows and columns of this column family are designed to support reads in a highly parallel fashion and the limit of 20K tweets per row ensures

that no row goes above the recommended maximum size for a given row. All of these features combine to make the new design highly scalable and able to support efficient data analysis.

The Event_Information column family is designed to keep track of metrics that would otherwise take a long time to compute. Furthermore, the design of this table supports this goal while also making it easy to update its various columns while data collection is occurring. For instance, the `et_rowkeys` column can easily be updated each time a new row key is created by simply appending the new row key to the end of its array. The same thing can be done for the `keywords` column whenever a new keyword is added to the data collection and likewise each day that a collection is active will cause a new Julian date to be added to the `julian_dates` column. Furthermore, these columns provide a big picture view of the dataset allowing keywords, all generated row keys, and the days the collection was active to be retrieved with a single API call. In the EPIC Collect original data model, all of this information, except for an event's keywords, had to be computed by iterating over the rows of the Filter_Tweet column family; keywords could be looked up directly in the Event_Filter column family of the original design. Now, this information can be retrieved in constant time and the process of row key generation described above in Chapter 7 has been eliminated. These benefits serve to speed up data analysis by avoiding unnecessary calculations and row/column traversal.

The Event_Abstractions column family plays a key role in being able to provide efficient batch data analytics. For each collected event, it creates three indexes that allow the most commonly requested queries of Project EPIC analysts to be answered quickly with just a few API calls: retrieve all tweets that contain a particular keyword; retrieve all tweets that are geotagged; and retrieve all tweets that were generated on a particular day or set of days. In addition, date ranges can be applied to the keyword and geotagged queries to allow for fine

grained queries that also complete quickly. As mentioned above, this column family can easily be extended with additional abstractions by simply adding new rows that index the tweets in the Event_Tweets column family in some new way, such as the indexes that our incremental sorting method creates to allow large datasets to be sorted along a number of dimensions [Aydin and Anderson 2015]. All of these indexes increase Cassandra’s ability to support a wider range of data analysis tasks while minimizing the number of external tools (such as Solr) that need to be used alongside Cassandra to support arbitrary queries over the collected datasets.

Furthermore, the IDCA App was developed to efficiently make use of the IDCAP. The IDCA App provides a well-designed user interface for analysts to orchestrate the IDCAP to stream, persist, and monitor tweets in real-time. The IDCA App provides incremental analytics that allows analysts to perform their queries in real-time at interactive speeds without worrying about big data collection and analytics related challenges.

To conclude, developing the IDCAP requires software engineering skills to trade one class of technology for another, and it involves determining the proper architectural style that supports data analytics using both real-time and batch processing techniques. In particular, this thesis identified ways for Project EPIC’s data intensive systems—EPIC Collect and EPIC Analyze—to significantly improve existing features and offer new capabilities to Project EPIC analysts, primarily by shifting their orientation from a batch-oriented approach to one that enables real-time data analysis of active crisis events. Although the IDCAP’s new data modeling and column family design was applied to Twitter datasets and the domain of crisis informatics, our design techniques can be applied to other application domains and datasets to provide scalable and incremental storage and analytics more broadly. Therefore, this thesis represents a contribution to software engineering with respect to

- software architecture design and technology trade-offs,
- a prototype infrastructure for transitioning from batch data processing to real-time data collection and analytics
- techniques and methods for proper data modeling of large data sets and column family design techniques that other software engineering researchers can use in their own work when making use of columnar NoSQL data stores.

BIBLIOGRAPHY

Aaron Schram and Kenneth M. Anderson. 2012. MySQL to NoSQL: Data Modeling Challenges in Supporting Scalability. *In ACM Conference on Systems, Programming, Languages and Applications: Software for Humanity*. 191–202.

Ahmet Arif Aydin and Gita Alaghband. 2013. Sequential and parallel hybrid approach for non-recursive most significant digit radix sort. *In 10th International Conference on Applied Computing*. 51–58.

Ahmet Arif Aydin and Kenneth M. Anderson. 2015. Incremental Sorting for Large Dynamic Data Sets. *In 2015 IEEE First International Conference on Big Data Computing Service and Applications*. 170–175.

Apache Cassandra. 2016. [Online]. Available: <http://cassandra.apache.org/>. [Accessed: 12-May-2016].

Apache Pig. 2016. [Online]. Available: <https://pig.apache.org/>. [Accessed: 12-May-2016].

Apache Solr. 2016. [Online]. Available: <http://lucene.apache.org/solr/>. [Accessed: 12-May-2016].

Apache Spark. 2016. [Online]. Available: <http://spark.apache.org/>. [Accessed: 12-May-2016].

Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review* 44, 2 (April 2010), 35–40.

Casey McTaggart. 2012. Analysis and Implementation of Software Tools to Support Research in Crisis Informatics. Master's thesis. University of Colorado Boulder, CO, USA.

Carl Hinsman, Neeraj Sangal, and Judith Stafford. 2009. Achieving agility through architecture visibility. *In 5th International Conference on the Quality of Software Architectures*. 1–16.

David Garlan and Mary Shaw. 1993. An Introduction to Software Architecture. *Advances in Software Engineering and Knowledge Engineering*, vol. I, V. Ambriola and Tortora, Eds. New Jersey: World Scientific Publishing Company.

David Garlan. 2000. Software architecture: a roadmap. *In Proceedings of the Conference on The Future of Software Engineering*. 91–101.

DataStax. 2016. [Online]. Available: <https://docs.datastax.com/>. [Accessed: 12-May-2016].

DataStax Enterprise. 2016. [Online]. Available: <http://www.datastax.com/products/datastax-enterprise>. [Accessed: 12-May-2016].

Debin Jia, Zhengwei Liu, Xiaoyan Gu, BoLi, Jingzi Gu, Weiping Wang, and Dan Meng. 2015. LuBase: Search-Efficient Hybrid Storage System for Massive Text Data. *In 15th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP 2015)*. 134-148.

Dewayne Perry and Alexander Wolf. 1992. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes* 17, 4(October 1992), 40-52.

Han Hu, Yonggang Wen, Tat-Seng Chua, and Xuelong Li. 2014. Toward Scalable Systems for Big Data Analytics: A Technology Tutorial. *In IEEE Access* 2, 652–687.

HBase. 2016. [Online]. Available: <http://hbase.apache.org/>. [Accessed: 12-May-2016].

HDFS. 2016. [Online]. Available: <http://hortonworks.com/apache/hdfs/>. [Accessed: 12-May-2016].

Hemant Purohit, and Amit Sheth. 2013. Twitris v3: From Citizen Sensing to Analysis, Coordination, and Action. *In Proceedings of 7th AAAI Conference on Weblogs and Social Media*. 746–747.

Jan Bosch. 2004. Software architecture: The next step. In first European Workshop (EWSA 2004). 194–199.

Jean-Daniel Fekete. 2013. Visual Analytics Infrastructures: From Data Management to Exploration. *IEEE Computer* 46, 7(March 2013). 22–29.

Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Communications of ACM* 51, 1 (January 2008), 107–113.

Kate Starbird, Leysia Palen, Amanda L. Hughes, and Sarah Vieweg. 2010. Chatter on the Red: What Hazards Threat Reveals About the Social Life of Microblogged Information. In *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work*. 241–250.

Kenneth M. Anderson. 2015. Embrace the Challenges: Software Engineering in a Big Data World. In *2015 IEEE/ACM 1st International Workshop on Big Data Software Engineering*. 19–25.

Kenneth M. Anderson and Aaron Schram. 2011. Design and Implementation of a Data Analytics Infrastructure in Support of Crisis Informatics Research (NIER track). In *33rd International Conference on Software Engineering*. 844–847.

Kenneth M. Anderson, Aaron Schram, Ali Alzabarah, and Leysia Palen. 2013. Architectural implications of social media analytics in support of crisis informatics research. *IEEE Bulletin of the Technical Committee on Data Engineering* 36, 3 (September 2013), 13–20.

Kenneth M. Anderson, Ahmet Arif Aydin, Mario Barrenechea, Adam Cardenas, Mazin Hakeem, and Sahar Jambi. 2015. Design Challenges/Solutions for Environments Supporting the Analysis of Social Media Data in Crisis Informatics Research. In *48th Hawaii International Conference on System Sciences*. 163–172.

Leysia Palen, Jim Martin, Kenneth M. Anderson, and Douglas Sicker. 2009. Widescale Computer-Mediated Communication in Crisis Response: Roles, Trust & Accuracy in the Social Distribution of Information. (2009)
<http://www.nsf.gov/awardsearch/showAward.do?AwardNumber=0910586>

Leysia Palen, Kenneth M. Anderson, Gloria Mark, James Martin, Douglas Sicker, Martha Palmer, and Dirk Grunwald. 2010. A Vision for Technology-Mediated Support for Public Participation & Assistance in Mass Emergencies & Disasters. In *Proceedings of the 2010 ACM-BCS Visions of Computer Science Conference*. Article 8. 12 pages.

Leysia Palen, Sarah Vieweg, and Kenneth M. Anderson. 2011. Supporting ‘Everyday Analysts’ in Safety and Time-Critical Situations. *The Information Society* 27, 1 (January 2011), 52–62.

Liming Zhu, Len Bass, and Xiwei Xu. 2012. Data Management Requirements for a Knowledge Discovery Platform. In *Joint 10th Working IEEE/IFIP Conference on Software Architecture & 6th European Conference on Software Architecture*. 169–172.

Mario Barrenechea, Kenneth M. Anderson, Ahmet Arif Aydin, Mazin Hakeem, and Sahar Jambi. 2015. Getting the Query Right: User Interface Design of Analysis Platforms for Crisis Research. In *2015 International Conference on Web Engineering*. 547–564.

Mark A. Cameron, Robert Power, Bella Robinson, and Jie Yin. 2012. Emergency situation awareness from twitter for crisis management. In *Proceedings of the 21st International Conference Companion on World Wide Web (WWW '12 Companion)*, 695-698.

Mauro Andreolini, Michele Colajanni, and Stefania Tosi. 2011. A software architecture for the analysis of large sets of data streams in cloud infrastructures. In *11th IEEE International Conference on Computer and Information Technology*. 389–394.

MongoDB. 2016. [Online]. Available: <http://www.mongodb.org/>. [Accessed: 12-May- 2016].

Mourad C. Oussalah, F. Bhat, Kate Challis, and Thorsten Schnier. 2013. A Software Architecture for Twitter Collection, Search and Geolocation Services. *Knowledge-Based Systems* 37 (January 2013), 105–120.

Michael J. Mior. 2014. Automated schema design for NoSQL databases. In *Proceedings of the 2014 SIGMOD PhD Symposium*. 41–45.

Michael Mattsson, Hakan Grahm, and Frans Mårtensson. 2006. Software architecture evaluation methods for performance, maintainability, testability, and portability. ... Qual. Softw. ..., 2006.

Mike Barlow. 2013. Real-Time Big Data Analytics: Emerging Architecture (1st. ed.). *OReilly Media*, Sebastopol, CA.

MySQL. 2016. [Online]. Available: <https://www.mysql.com/>. [Accessed: 11-May-2016].

RabbitMQ. 2016 [Online]. Available: <http://www.rabbitmq.com/>. [Accessed: 12-May-2016].

Redis. 2016. [Online]. Available: <http://redis.io>. [Accessed: 12-May-2016].

Scott Jarr. 2015. *Fast Data and The New Enterprise Data Architecture* (1st. ed.). *OReilly Media*, Sebastopol, CA.

Sudha Verma, Sarah Vieweg, William J. Corvey, Leysia Palen, James H. Martin, Martha Palmer, Aaron Schram, and Kenneth M. Anderson. 2011. NLP to the Rescue? Extracting Situational Awareness Tweets During Mass Emergency. *In Fifth International AAAI Conference on Weblogs and Social Media*. 17–21.

Xiang Li, Zhiyi Ma, Hongjie Chen. QODM: A query-oriented data modeling approach for NoSQL databases. 2014. *In IEEE Workshop on Advanced Research and Technology in Industry Applications*. 338–345.

Xiaoming Gao and J. Qiu. 2014. Supporting Queries and Analyses of Large-Scale Social Media Data with Customizable and Scalable Indexing Techniques over NoSQL Databases. *In 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 587–590.

The Search API. 2016. [Online]. Available: <https://dev.twitter.com/rest/public/search>. [Accessed: 12-May-2016].

