A Software Architectural Design for Automated Data Processing in Data-Intensive Software Systems

by

Mazin Hakeem

B.S., King Abdulaziz University, 2007

M.S., University of Colorado at Boulder, 2012

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2019

This thesis entitled:
A Software Architectural Design for Automated
Data Processing in Data-Intensive Software Systems
written by Mazin Hakeem
has been approved for the Department of Computer Science

Prof. Kenneth M. Anderson

Prof. Christine Lv

Prof. Shivakant Mishra

Prof. Tom Yeh

Prof. Stephen Voida

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.

Hakeem, Mazin (Ph.D., Computer Science)

A Software Architectural Design for Automated

Data Processing in Data-Intensive Software Systems

Thesis directed by Prof. Kenneth M. Anderson

Data-intensive software systems allow analysts to investigate large data sets, consisting of billions of elements, that would otherwise be impossible to explore in a systematic fashion. These systems make use of clusters of machines and deploy a variety of software services to provide scalability, availability, and reliability to ensure that analysts can import new data, issue potentially long-running queries, and receive answers reasonably fast for the scale involved. Support for automated analysis is often lacking, however, placing a burden on analysts to keep track of what analysis techniques have been applied to which data sets; where—in a possibly complicated analysis workflow—each data set is; and how much further does it have to go before all questions concerning a data set have been answered.

In this thesis, I focus on the design, development, and deployment of a set of software services that will help data-intensive software systems address these issues by automating the application of analysis techniques to collected data sets, whether they are actively being collected or are archived historical data. I implement the design of these services within the context of an existing data-intensive software system known as the Project EPIC software infrastructure; this system supports crisis informatics research based on the collection and analysis of large Twitter data sets generated during mass emergency situations. In this document, I present the design of my services and the software architecture that supports their integration into data-intensive software systems. I also conduct a performance evaluation of the software prototype that implements my design and report on its results. I conclude by reflecting on the benefits my approach to the design of my software infrastructure has on software engineering and crisis informatics research.

Acknowledgements

I would like to start by thanking my advisor Professor Ken Anderson for his generosity and patience, for his endless support, and for his guidance in my PhD work all these years.

I want to thank my fabulous colleagues in the software engineering research group: Mario, Sahar, Adam, Rasha, Afnan, Ahmet, Reem, and Ali for their ongoing support.

Also, many thank to the Project EPIC team: Jennings, Melissa, Marina, Jo, Lise, and Robert for their support and kindness.

I am also grateful for my fellow PhD students: Darren, Kyle, Khalid, Ehab, Halley, Anshul, Mike V., Mike & Jackie, Daniel, Hooman, and Annika for their friendship and for keeping me humble. The list is too big, so sorry for missing out on others.

I am thankful to King Abdulaziz University for granting me a scholarship to fulfill my academic journey.

Finally, I thank my siblings and my late mother, for their unconditional support, love, patience and care while being so far away from them for all these years.

Contents

Tables

Table

Figures

Chapter 1

Introduction

Modern computing environments are producing massive amounts of data each day thanks to the emergence of different kinds of internet-based applications. Social media platforms, in particular, play host to a diverse set of users who generate millions of posts that are shared either publicly or privately. The content of these posts range from entertainment, the latest news about celebrities, jokes about politicians and world leaders, links to online tutorials, discussions about which smartphone is supreme, and more. These posts can be analyzed to extract useful and interesting information about the people sharing them. For example, studying the content related to certain pop culture topics or celebrity news can provide insight about the age range of those sharing that information. Studying posts about prominent sports teams during a season can give ad agencies demographic information so they can prepare marketing strategies.

While mining information for marketing purposes has its use, the field of *crisis informatics* [49, 47], has shown that these same social media platforms become a home for information sharing and collaboration around mass emergency events, such as natural disasters. Researchers in this field will collect social media posts generated during a mass emergency to study the collaboration behaviors and information dissemination patterns that occur in these events, to provide useful information to emergency responders, and to provide guidelines to the various populations that converge on the event digitally on how to use these platforms to help the impacted area and the people who live there.

A multidisciplinary research group at the University of Colorado Boulder—known as Project

EPIC—has been conducting crisis informatics research since Fall 2009 and, indeed, Project EPIC's principal investigator, Prof. Leysia Palen, helped to create the field with her initial work that started at CU Boulder in 2005. Over the intervening years, Project EPIC has developed software systems to collect and analyze large sets of social media data, known as EPIC Collect [54] and EPIC Analyze [2, 7]. However, the initial design of these systems mostly supported batch processing to analyze the collected data; this approach forces researchers to wait until an event is finished before complete analysis can commence. Some work has since occurred to augment the batch processing features of EPIC Analyze with real-time analysis capabilities [5, 28]. However, these new systems were proof-of-concept prototypes that failed to address features that would make these new capabilities useful to crisis informatics researchers on a daily basis. For instance, these systems demonstrated that data could be collected in real-time and queried in efficient ways but dropped essential features that would allow analysts to annotate the collected data or to follow well-defined analysis workflows in which a newly collected data set needs to be "cleaned" and then processed by any number of analysis techniques to answer a common set of questions that analysts have about them.

For my dissertation, I address the features needed by crisis informatics researchers to ensure that the data-intensive software systems—also known as big data software systems—that they use to perform analysis can automate as much of the low-level data cleaning and processing as possible, allowing them to focus on their research questions for a particular event and designing new algorithms to help discover relevant information. A framework is also provided to plug new analysis techniques into the infrastructure; the new techniques will then automatically be applied to all collected data sets. The goal is to provide analysts with a framework that will allow them to turn repetitive, laborious analysis tasks into something that is automatically performed for them on all of the data sets that they create. Achieving this goal will free analysts to devote their time to more complex analysis tasks knowing that all straightforward tasks are being completed automatically.

To do this, I present a design with new capabilities for Project EPIC's software infrastructure; the new design provides mechanisms to automatically apply all available analysis techniques to all collected data sets, and to track the progress of each technique on each data set down to the level of an individual social media post. This tracking and management system is built to be scalable, extensible, and reliable, and it must be able to handle both the batch processing of previously collected data sets while also ensuring that all newly collected social media posts are processed by all available analysis techniques.

To achieve these goals, the design of my software prototype makes use of the microservices software architectural style. This style encourages an approach to the design of software services where each service is as modular, decoupled, and cohesive as possible such that each one performs a single function within the overall system, without interfering with other services. This approach will help to keep the code base small per service for easy maintenance and for each service to be independent [46, 63]. Such a paradigm is parsimonious with the use of cloud-based hosting services such as Google Cloud [6, 11] where each service can be packaged in a container, allowing each to be deployed and maintained independently. By embracing the microservices approach, the overall system supports extensibility such that services (and hence features) can be added or removed in a straightforward manner with only minimal impacts on the entire system.

The rest of my thesis document is organized as follows. Chapter 2 presents background information related to my thesis topic. Chapter 3 describes my approach and its limitations. I then present my research questions in Chapter 4. Chapter 5 lays out the design of my software infrastructure and presents its implementation as a software prototype. I then describe how I evaluated my work in Chapter 6 and present relevant related work in Chapter 7; Finally, I present various options for my future work in Chapter 8 and then conclude with a discussion of the implications of my work on software engineering and crisis informatics in Chapter 9.

Chapter 2

Background

In this section, I present information on topics that are relevant to my research questions. In particular, I present information on big data, big data software architecture, microservices, crisis informatics, and previous work performed by Project EPIC in the area of big data software systems.

2.1 Big Data

The Merriam-Webster online dictionary defines the term *big data* as "an accumulation of data that is too large and complex for processing by traditional database management tools" [45]. Big data has unique characteristics that cannot be handled by using a traditional SQL-based database management systems (DBMS). The data usually exceeds terabytes in size, and the format of the data is heterogeneous. These characteristics are commonly referred to as the *Four V's of Big Data*, where the V's are: volume, velocity, variety, and veracity. The following sections walk through the history of big data and the common properties (the V's) that distinguish big data from more traditional structured data sets.

2.1.1 A Brief History

The use of large amounts of data, or big data, was a little different in the past than what it is now [9]. Big data was needed for enterprises to store and query large historical business transactions for the purposes of analyzing them and generating reports. Such needs paved ways for research on "database machines" in the 1970s, which were dedicated machines designed with proprietary hardware and software to query and process data exclusively. Research in the 1980s led to developing parallel databases, such as the Gamma system [16], the GRACE system [23], and the Teradata system [55]. All of these systems provided declarative query languages for parallelizing storage and query processing. Teradata's database system designed for Kmart had a capacity of 1TB, which was considered to be large-scale at the time [9]. In addition to supporting big data, transaction processing (TP) systems emerged which empowered daily business activities to produce data in large volumes. In the 1980s, "big" was measured with transactions per second (TPS). 100 TPS was considered to be "fast" in 1985, while a SQL relational DBMS was able to score 200 TPS a couple of years later. These numbers are considered "mediocre" in comparison to today since modern TP systems can reach hundreds of thousands of TPS [9].

The 1990s started work on distributed systems and the World Wide Web (WWW). This was a time when big data became problematic, because the amount of data was growing rapidly and it was getting harder to index and query at the larger scales. Traditional databases could not fulfill the task, thus creating a challenge to manage the massive amounts of data being generated by web services and other applications. As a result, Google created the Google File System (GFS) [24] to handle storing large data distributed among commodity hardware. Then Google popularized the MapReduce programming model to process those massive amounts of data on its file system [15]. Open source alternatives such as Hadoop were developed to spread the benefits of this approach more broadly, which introduced the Hadoop File System (HDFS) [27] as an alternative to GFS. Hadoop of course allowed MapReduce jobs to run on its file system. Both Hadoop and MapReduce are popularly used for large-scale batch processing of big data [9].

2.1.2 The Four V's

The concept of big data has changed from what it used to be in the past, which mostly focused on the volume of the data. Currently, big data has four famous characteristics, which also represent challenges for software engineers, known as the "V's" of big data [31, 41, 42]. The first is *volume* which refers to the immense size of data in terabytes and beyond. The second is *velocity* which refers to the speed at which data arrives or is generated by some software system. An example would be the amount of tweets generated per second across Twitter's Streaming API, or the number of people visiting a website which generates a large number of log entries. The third is *variety* which refers to the different types and formats of data such as data represented in XML or JSON, or represented in different data structures like an array or hash table. The last one is *veracity* which is about the quality of the data, what value it provides, and whether it can be trusted (e.g. do we know where the data came from, do we trust the user who generated it, etc.). In Twitter, for example, tweets can be used to spread misinformation such as the "fake news" phenomenon that came to prominence in the 2016 U.S. Presidential election [59].

2.2 Big Data Processing Architectures

With the emergence of NoSQL databases that support the storing of massive data sets, a number of technologies have been created to support efficiently processing them, such as Hadoop [27], Apache Storm [61], Spark [58], and many others. These technologies are known as distributed computation frameworks since their abstractions make it straightforward to specify computations that take place across thousands of machines simultaneously. These platforms make it possible to analyze data sets that otherwise could not be processed (indeed, even stored) by a single machine. I now present two software architectures that have emerged to take advantage of distributed computation frameworks and NoSQL databases; these architectural styles provide guidance to software engineers designing big data software systems.

2.2.1 Lambda Architecture

The Lambda Architecture is a big data processing architecture designed by Nathan Marz. He defines it as a "general-purpose approach to implementing an arbitrary function on an arbitrary data set and having the function return its results with low latency" [43]. The architecture is made of three layers (as shown in Fig. 2.1)—the batch layer, speed layer, and serving layer—accessed via queries submitted to a Query subsystem.

The goal of this architecture is to support the design of fault-tolerant and distributed data processing systems that are also able to process large amounts of data in real-time and via batch processing. The design of the Lambda architecture allows ad-hoc data querying on arbitrary data sets with as low latency as possible. Additionally, the architectural design enables integrating and using multiple kinds of big data technologies such as NoSQL databases and distributed messaging systems for storing and queuing data respectively. These new big data technologies are built to handle fault-tolerance, resilience, sharding, and many other distributed system related features relieving developers from adding and managing those features in their systems, something that was difficult to achieve with traditional databases [43].

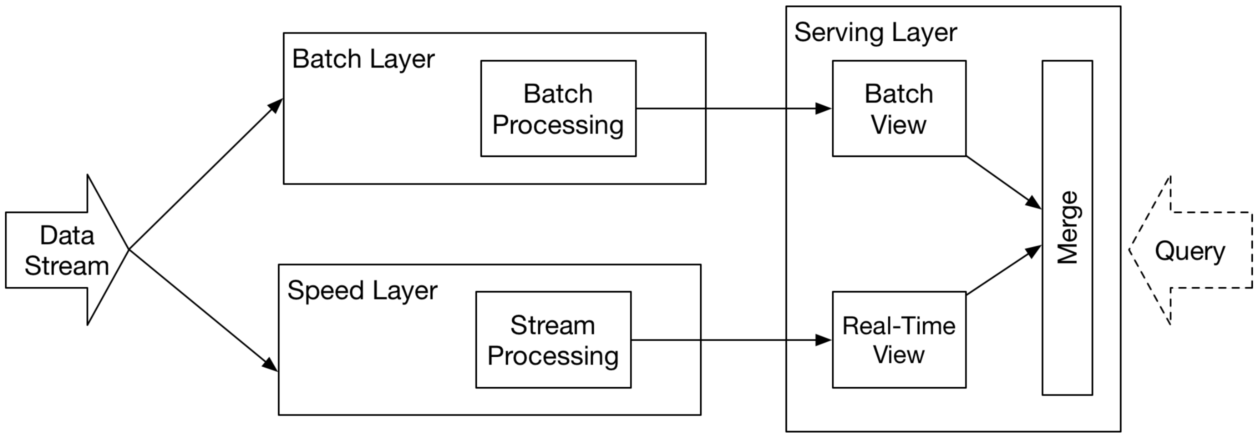


Figure 2.1: The Lambda Architecture [43]

In the batch layer, data is stored in its original format where all new data is appended to an existing storage mechanism, thus making the storage mechanism immutable. Another functionality of this layer is to compute all downstream data by running batch processing jobs via MapReduce. The results of these jobs are called "batch views." The speed layer handles incremental processing of the incoming data as it arrives from the source. This layer is concerned with processing only the data it has at the moment, which allows it to compute a small set of downstream data in real time, called the "real time views." In other words, it provides a real time view of new data. The serving layer is responsible for indexing the "batch views" generated from the batch layer so that the views can be queried. Due to indexing, it provides fast random reads of the batch processed data. The last part of the architecture is called "Query." Here, ad hoc questions can be answered by merging the results from the batch views and the real-time views.

2.2.2 Kappa Architecture

The Kappa Architecture (Fig. 2.2) is an architectural design for data processing suggested by Jay Kreps, co-founder and CEO of Confluent, and principal staff engineer at LinkedIn [34]. It is considered a more simplified version of the Lambda Architecture where the Batch Layer has no place in the new design [33]. In the Lambda Architecture, the Batch Layer stores the data in its original format in an append-only immutable fashion. To replace this, the original data is logged in a queue or log system such as Kafka instead.

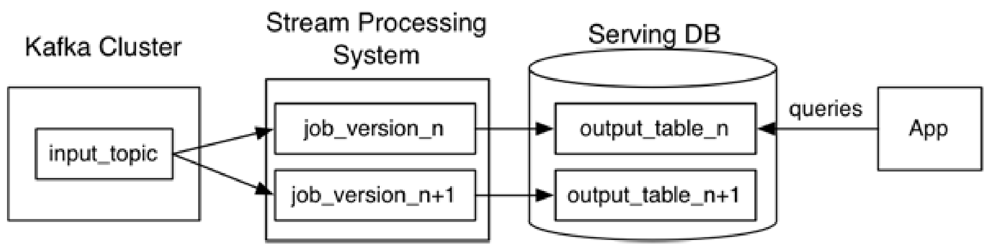


Figure 2.2: The Kappa Architecture [34]

The data is then streamed for processing or reprocessing by using data streaming technologies like Apache Storm or Spark. The processed output is saved to a database to be queried by users. If the data needs to be reprocessed when the processing code changes or gets improved, another processing instance would stream the logged data from the original source. The reprocessed data is stored in another table in the database and the old one is deleted once the new reprocessing is fully finished and stored.

The idea behind Kappa is to maintain only one set of code and change it when needed, unlike the situation with the Lambda Architecture where separate code is needed for the batch and speed layers [34]. The goal is to also permanently store the incrementally processed data such that it can be later viewed in parts, or as a single whole once everything is processed. According to Kreps [34], Kappa has served as the basis to design and implement production software systems [53].

2.3 Microservices

The microservices software architectural style was popularized by Fowler and Lewis [22]; it promotes decomposing a software system into small, modular, decoupled services with single responsibilities. These services are independently deployable and maintained. They run in their own separate processes and they expose their functionalities for intercommunication through lightweight HTTP-based APIs like REST [6, 30, 63].

2.3.1 Building modular services before microservices

Earlier technologies like CORBA [12], DCOM [14], and J2EE [25] attempted to decompose software systems via software components to help promote reusability. Unfortunately, developers encountered difficulties building systems from these technologies since each component technology made use of a different communication protocol leading to complex interoperability problems [30]. Service-oriented architecture (SOA) [50] tried to fix the interoperability problems by setting standards for building web services after the XML data format and the HTTP protocol appeared. However, SOA enforced restrictions to make use of an overwhelming number of standard protocols all at once [22], such as WSDL (a description language for defining services), SOAP (an XML based RPC communication protocol), and the Enterprise Service Bus or ESB (a layer that has the logic for “service orchestration” which also had its own standards) [50], just to name a few. The ESB introduced problems of its own by placing all communication logic into a big centralized layer [63]. The microservices architectural style does not enforce specific proprietary communication protocols, vendor specific middleware, or restrictions on specific technologies and standards as was done with SOA. An engineer can build multiple services using any technology stack that makes use of REST [21], or even SOAP, and use any messaging format like XML and JSON. Microservices is therefore considered a “specific approach to SOA” [46] or “SOA done right” [22].

2.3.2 Benefits

The idea behind microservices is that a whole system can be comprised of multiple small services. A service should perform one job and do it very well as an independent entity, and it should not be coupled with other functionality. This decoupling allows the service to be easily managed and maintained. Thus, debugging an error or adding or improving features on that service should not have negative impacts on the larger software systems being created on a set of services.

Also, microservices can be scaled easily when needed for a specific reason. This helps to eliminate the technique of deploying multiple copies of a big monolithic system just to scale a certain feature. Assuming that one has multiple microservices and one of the services is performing poorly due to a lack of resources, only the poorly-performing service needs to be scaled independent of the rest of the system. With microservices, it is expected that each service is deployed independently.

In addition, if a single microservice fails for some reason, the chance that the whole system is impacted is greatly reduced. As a result, software systems that are constructed on top of microservices are expected to be resilient to failures providing partial functionality while waiting for a failed service to be restored. Lastly, microservices allow a software system to be composed of multiple microservices that are built using various technologies, frameworks, and different programming languages, without relying on a single technology stack. This empowers developers to select the frameworks that best fit the needs of an individual service [30] while allowing the architecture to be more flexible and heterogeneous [46].

2.3.3 Challenges

Although building microservices promotes modularity and resilience, these benefits come with a price. A team of developers needs to be skilled or comfortable with distributed systems concepts [6] since building microservices and allowing each service to intercommunicate via REST or via message queues adds complexity to the software development effort. Also, software engineers that design systems in this way will need operational knowledge of system administration skills—sometimes referred to as DevOps—to manage, deploy, provision, and monitor the services.

2.4 Crisis Informatics

Crisis Informatics is a multidisciplinary research area that studies how the public communicates, reacts, and responds when mass emergency events take place by using social media platforms like Twitter [1, 2, 48]. The goal is to understand how these new communication platforms impact both the formal and informal response to these events. Emergency responders and other government officials need to understand how to use these platforms to extract situational awareness and to provide timely information about the crisis. Members of the public can use these platforms to publish information about the crisis, seek information about the crisis, and/or collaborate with others to help those in need. Finally, crisis informatics researchers want to understand how these behaviors so they can be used to drive the design of new software services that people can use during these future emergencies.

The researchers in this area need to investigate the data that is produced from social media platforms and the amount of that data can be very large, full of noise, and available in various formats, such JSON, XML, or CSV. The main challenge in supporting such research is to provide a repository to store the massive amounts of data collection in its original format while the disaster is happening. That will allow the researchers to have access to the data to study at a later time. The other challenge is to provide the means to analyze the large collection of data and to sift through the collection in manageable chunks to aid in discovering useful insights.

2.5 Project EPIC Software Systems

Project EPIC (Empowering the Public with Information in Crisis) is a multidisciplinary research group that focuses on studying crisis informatics. To support Project EPIC researchers and analysts who study the public data generated when a mass emergency occurs, a few systems were designed and built to support the collection of massive amounts of data and to aid in managing and analyzing that data.

2.5.1 EPIC Collect

EPIC Collect is a system that collects social media data during crisis events to support crisis informatics research [3]. EPIC Collect was built to be reliable, fault tolerant, scalable, and highly available. Indeed, it has been running 24/7 in some fashion since January 2010 and in its current form since the Summer of 2012 [54].

The data source of choice for Project EPIC's crisis informatics researchers was Twitter data. Thus, the collection software relies on Twitter's Streaming API to stream tweets that match a list of keywords that represent terms related to the current set of crisis events under study. These collected tweets are then stored in a 4-node Cassandra cluster (Fig. 2.3) due to Casandra's capabilities to efficiently write and store terabytes of data. Cassandra also automatically replicates and shards the data assuring that no data is lost and that the database is always available [39].

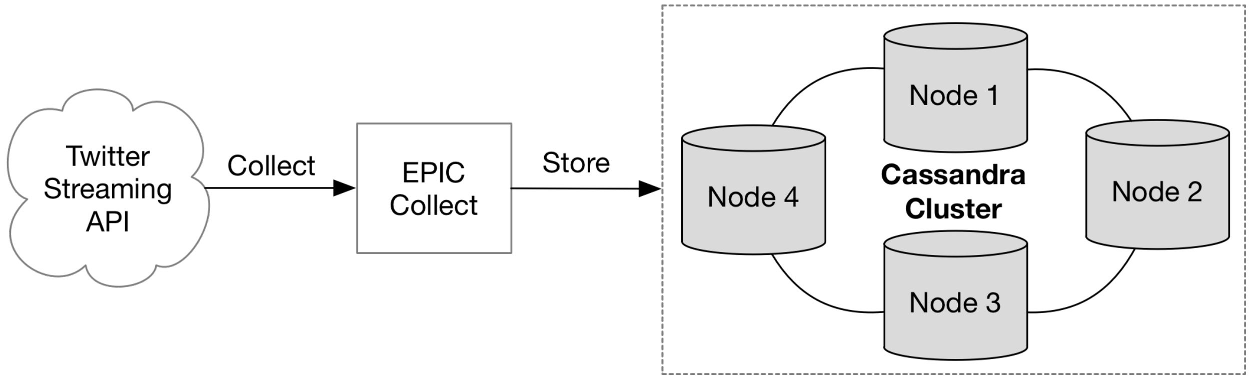


Figure 2.3: EPIC Collect

To use the system, when a mass emergency event takes place, analysts observe Twitter to curate a comprehensive list of relevant keywords. They then access a web application called the Event Editor to create a new event with the list of keywords to start a new collection. The current list of keywords (spanning multiple events) is provided to Twitter's Streaming API, and the collected tweets are then received, classified, and stored. The system has collected more than

2.5 billion tweets for hundreds of events since it started collecting in 2010, consuming terabytes of disk space.

2.5.2 EPIC Analyze

EPIC Analyze is a web-based application that allows analysts to browse, search, annotate, and visualize large Twitter data sets that were collected for numerous natural disaster events [2, 7]. Prior to this web application, EPIC Collect was available for collecting tweets as mentioned above but provided no support for analyzing them. To work on the collected tweets, a specific program was written to extract data from EPIC Collect and convert it to a different format for analysts to use such as CSV or JSON. They then used analytics tools such as MS Excel or Tableau to help them study the data. Most third party analytics applications were not built to consume gigabytes of data. Thus, the analysts had to filter and narrow down the data which required either some programming expertise or working with a developer. The whole process was ad hoc and time consuming for the analysts.

Prior to EPIC Analyze, a prototype was built to allow analysts to explore collected tweets [44]. McTaggart elicited requirements for building a Twitter analytics system by conducting a usability study, and by interviewing analysts to provide various kinds of analytics features to answer common data exploratory questions. It was one of the first systems to provide analytics for Project EPIC. Her prototype was then used alongside Excel and other third party applications by the analysts as mentioned earlier to run ad hoc sampling, filtering, sorting, and analyzing collected tweets. However, she reported that her prototype could not scale to larger data sets, and, as a result, slow performance issues would occur leading to delays that made the software inappropriate for daily use.

EPIC Analyze is an extension of EPIC Collect which accommodates the needs of analysts to filter the tweets, view them in a readable format, annotate the tweets for downstream analysis, visualize them in a graph to display high Twitter activity, and fine tune the search results to narrow down the amount of tweets that need to be studied. To provide all the necessary features to the an-

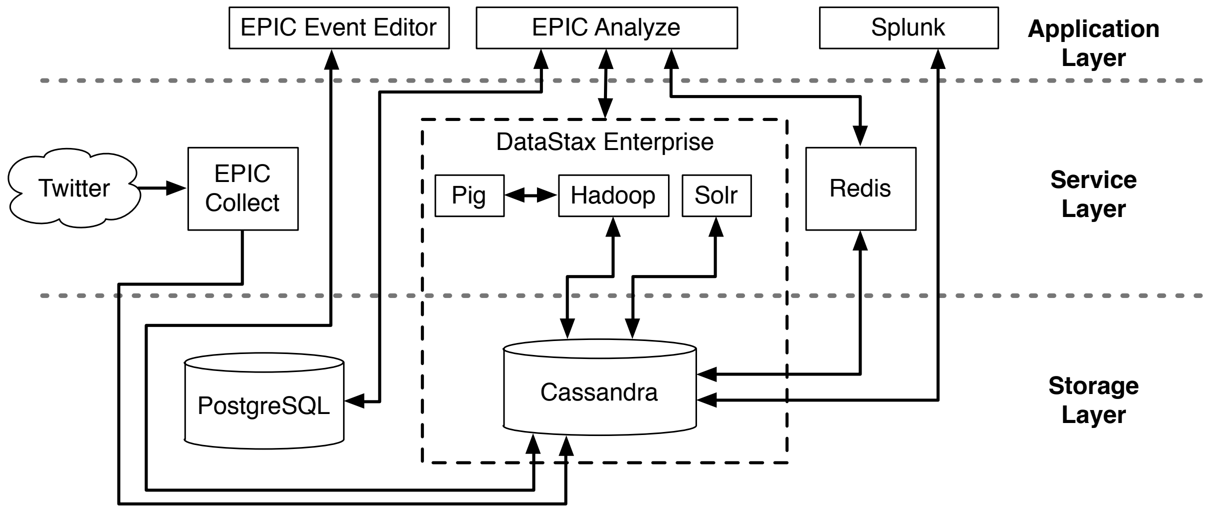


Figure 2.4: EPIC Architecture Diagram

alysts for a seamless fast experience, EPIC Analyze's architecture was designed for "scalability and performance" and to be heterogeneous [2]. Therefore, the design embraces a layered architecture to scale the application easily and to add other features and services when needed. For heterogeneity, the infrastructure takes advantage of DataStax Enterprise [13] which is an amalgamation of multiple NoSQL and big data analytics technologies such as Cassandra, Solr, Spark, Hadoop, and others. These tools in the DataStax edition are well integrated to take the burden away from developers to glue these technologies together to effectively use them.

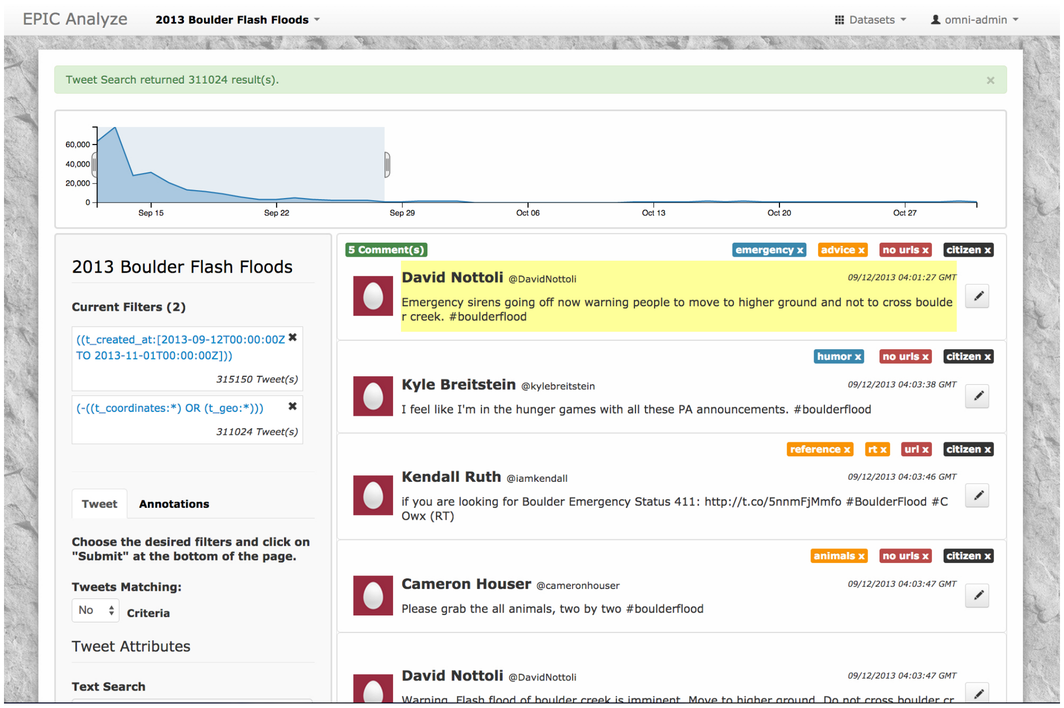
As mentioned above, EPIC Analyze's approach to data processing is batch-oriented; that is, an event's data set needs to be batch indexed ahead of time before the data set can be viewed and/or filtered. The batch indexing is performed by extracting data from EPIC Collect and then feeding it to Solr [57], an open source NoSQL based search technology that uses Lucene to index text data. Once indexed, the analysts can view and analyze the data using EPIC Analyze's web interface which is built with Ruby on Rails. The users can sift through millions of tweets without any delays thanks to Solr's indexing and Redis's caching mechanism that allows fast pagination of tweets. Also, Postgres is used to keep track of annotations stored on specific tweets; if a search query involves annotations and other Tweet attributes, Postgres can be used to optimize the search performed by Solr by supplying the tweet ids of tweets that match the annotation part of the query. Since the number of tweets that have been annotated is very small in relation to the total number of tweets in the system, this optimization provides a significant speed up for this type of query [2].

The integration of EPIC Collect and EPIC Analyze was not as straightforward as Fig. 2.4 conveys. In particular, EPIC Collect stores tweets in a Cassandra cluster that is distinct from the Cassandra cluster that is used by EPIC Analyze to index tweets. Not shown in Fig. 2.4 is the automated import process that occurs to bring tweets from EPIC Collect's Cassandra cluster to the one used by EPIC Analyze. During that process, tweets are re-formatted and placed in a special column family that can be automatically indexed by Solr. EPIC Analyze makes use of Datastax Enterprise [13]; that software package provides special versions of Solr and Hadoop that can read/write information stored in Cassandra.

When Project EPIC first imported tweets from EPIC Collects Cassandra cluster, the migration failed since that version of Cassandra was making use of a schema built using Thrift, which was no longer supported by Datastax Enterprise [62, 40]. Instead, the newer versions of Cassandra made use of schemas that can be processed by CQL (Cassandra Query Language), an SQL-like syntax; furthermore, the DataStax Enterprise version of Solr would only index data whose schema was built using CQL. Once Project EPIC did the work of implementing a suitable migration process, Twitter data sets could be imported into EPIC Analyze and then automatically indexed by Solr. The benefit of all this work was that EPIC Analyze could then support full-text search on its collected data sets. I present this information to highlight the challenges encountered in big data software engineering.

Fig. 2.5 shows the web interface for EPIC Analyze where an analyst can browse, search, and visualize Twitter data for a given event. At the top of the screenshot, an analyst can pick an event from a list to view (e.g. 2013 Boulder Flash Floods). Below that, an analyst can view general information about an event like the number of tweets it contains and a visualized timeline of the activity of Twitter users during that event. The right side shows individual tweets. That view is limited to fifty tweets per page; users muse click pagination links if they want to see more tweets that match the current query.

The small colorful tags that appear on some tweets are annotations, such as tags created by analysts to help categorize tweets. The left side of the page is where analysts can search and filter the data set based on a number of query parameters. In addition, a user can refine and narrow down the number of tweets to be viewed by chaining multiple queries together [7]. The query chaining feature enables each search query's returned result to be treated as a separate data set such that the new data set can be refined with more filtering, which will again return a new data set. The user can then continue to filter the data until he/she reaches a point where the number of queried tweets are manageable to study.



Tweet Search returned 311024 result(s).

2013 Boulder Flash Floods

Current Filters (2)

((t_created_at:[2013-09-12T00:00:00Z TO 2013-11-01T00:00:00Z]))
315150 Tweet(s)

(-((t_coordinates:*) OR (t_geo:*)))
311024 Tweet(s)

Choose the desired filters and click on "Submit" at the bottom of the page.

Tweets Matching:
No Criteria

Tweet Attributes

Text Search

5 Comment(s) emergency x advice x no urls x citizen x

David Nottoli @DavidNottoli 09/12/2013 04:01:27 GMT
Emergency sirens going off now warning people to move to higher ground and not to cross boulder creek. #boulderflood

humor x no urls x citizen x

Kyle Breitstein @kylebreitstein 09/12/2013 04:03:38 GMT
I feel like I'm in the hunger games with all these PA announcements. #boulderflood

reference x rt x url x citizen x

Kendall Ruth @iamkendall 09/12/2013 04:03:46 GMT
if you are looking for Boulder Emergency Status 411: http://t.co/5nnmFjMmfo #BoulderFlood #COwx (RT)

animals x no urls x citizen x

Cameron Houser @cameronhouser 09/12/2013 04:03:47 GMT
Please grab the all animals, two by two #boulderflood

David Nottoli @DavidNottoli 09/12/2013 04:03:47 GMT
Warning. Flash flood of boulder creek is imminent. Move to higher ground. Do not cross boulder cr

Figure 2.5: EPIC Analyze Screenshot

2.5.3 IDCAP

The Incremental Data Collection and Analytics Platform (IDCAP) [5] is a project designed by Ahmet Aydin in which he developed a framework of various big data technologies like Cassandra, Spark, Redis, and RabbitMQ to support an infrastructure that incrementally indexes and processes large amounts of incoming data for real time analytics and for efficiently storing that processed data for fast retrieval.

In his work, he addressed the shortcomings of EPIC Collect that forced developers and analysts to rely on batch processing to perform analytical experiments on the collected data. One of the problems was that the tweets were not stored efficiently in Cassandra although Cassandra can store gigabytes of data per rowkey. Arbitrary and unbalanced storage of data in each rowkey can cause major performance overhead, which was the case with EPIC Collect. Some rowkeys stored tweets that varied from a few hundred tweets to a few millions. This variation sometimes caused connection timeouts with Cassandra and slow retrieval of data for some events. Also, there were duplicated tweets since the rowkey is defined with a keyword, julian date, and a random hexadecimal digit (`keyword:juliandate:[0-f]`). This creates a possibility where a tweet collected on a given day can be stored multiple times in different rowkeys if that tweet has multiple keywords that are being collected. For example, if we are collecting on keywords such as “floods” and “boulder,” we will have two rowkeys that are storing tweets with these keywords. If there is a tweet that contains both of these keywords, it will be stored twice.

The other problem was that it took some time to get simple information about an event such as the amount of tweets collected, the number of geotagged tweets, and many other insights that are easily accessible in relational databases. Since Cassandra does not have common relational DB features because it only indexes its rowkeys, and since EPIC Collect was only designed for collection, not analysis, answering these simple questions required batch processing. This problem was addressed in EPIC Analyze, but it still used batch processing, as discussed above.

To address these two issues in IDCAP, a new data model was designed to ensure no duplication of data, uniform data storage, and balanced data storage by following DataStax's recommendations for the size of data that should be stored in a wide rowkey. Their recommendation is 100 MB per row which yields about 20,000 tweets. Once this limit is reached, a new rowkey is created and new tweets are added accordingly, and the last digit in the rowkey is incremented for an event that is being collected (`event_a:1, event_a:2, ...`). This revised data model helps with fast reads and writes since each rowkey has a well-known maximum limit.

Furthermore, the new data model provides access to a "big picture" of an event which helps in obtaining information like the amounts of tweets for an event, the number of keywords of an event and the number of tweets per keyword, how many rowkeys are there for an event, the amount of geotagged tweets and on which day, and more. All these insights are incrementally indexed and processed while the tweets are being collected. This gives the analysts a better idea of what the general state of an event is in real-time without relying on batch processing tweets after the fact.

Lastly, a web application was built for IDCAP to start a data collection and to provide a near-real-time analytic view of the tweets being collected. This view includes a world map that shows how many tweets were tweeted from a certain location, a pie chart that views the amount of tweets that were collected for each day for an event, and a second pie chart that demonstrates the keywords and the percentage of tweets that contain them for an event that is being collected.

2.5.4 EPIC Real-Time

EPIC Real-Time [28] is a microservices-based big data analytics system that allows Project EPIC's analysts to investigate Twitter data in real-time. It does this by running a wide range of queries on real-time streaming data and on historical data at the same time. The system adopts elements of the Lambda Architecture to process data in batch and real-time, and then aggregates the results into a single view for the user. In addition, EPIC Real-Time is designed and implemented using reactive programming techniques to provide many important features of data-intensive software systems such as scalability, reliability, and extensibility. EPIC Real-Time's software architecture relies on multiple technologies such as Kafka, Cassandra, Spark, and a library based on the reactive programming paradigm called Akka.

The motivation for EPIC Real-Time was to address a problem the Project EPIC analysts had with discovering insights in large data sets for crisis informatics research. A user study was conducted by interviewing Project EPIC analysts to understand the most common research activities they practice to study and investigate collected Twitter data. This study helped identify the most common set of metrics and analytics that the analysts would use to find answers to their questions. This resulted in defining three types of queries for EPIC Real-Time: data set, trend, and popular. Firstly, the data set query, which is computed by default on all events, provides a handful of analytics insights that are related to the uniqueness of a certain field in an event such as a list of unique users, unique locations, unique URLs, and unique mentions. Secondly, the trend query provides information related to high frequency items in an event, such as the most tweeted (trending) hashtags, most mentioned users, highly active users, and most posted URLs. Lastly, the popular query is similar to the trend query but focuses on popularity such as the most retweeted tweets and most retweeted users.

The system was composed of two REST-based services: the Events service and the Query Engine. The Events service is used for creating events to collect data by injecting a list of keywords to Twitter's Streaming API, and then persisting the collection in Cassandra. The Query Engine allows a user to initiate different types of predefined queries.

As mentioned earlier, EPIC Real-Time follows the structures of the Lambda architecture, which is composed of three layers: batch, speed, and serving. The batch layer is responsible for storing data in its original format, batch processing that data, and persisting the processed data. The speed layer handles real-time data processing in which the data is streamed and incrementally processed. The serving layer is responsible for merging the results from both layers and revealing the outcome to a web application. When a query is submitted to explore an event's data, the batch layer begins batch processing by initiating Spark jobs to process historical data. The calculation is performed on the data based on metrics in a query such as the data set query, and the results are saved in Cassandra. The speed layer initiates a Spark job similar to the batch layer, but it focuses on processing the most recently collected data (e.g. the last five seconds). In the serving layer, the processed data from both layers is merged; it then exposes the results via a REST endpoint.

The final component of EPIC Real-Time is a web application that calls the REST APIs of the two services. The users can access the web application and start collecting Twitter data by providing a list of keywords. While the data is being collected, a query can be issued and metrics like the most tweeted hashtags are displayed and updated in real-time.

Chapter 3

Current Limitations and Approach

EPIC Collect has proven over time to be a reliable Twitter data collection infrastructure; its software architecture was designed to keep the system running 24/7 with high reliability. However, its purpose was only to store data reliably, and thus it provided no support for helping analysts study the data sets it collected. Accessing data stored within EPIC Collect is tedious for analysts, and they would prefer to have easy and direct access to the data. Instead, they had to rely on software engineers to run ad hoc analysis or to extract and reformat the data so that it could be loaded into applications often used by the analysts.

To address this situation, EPIC Analyze was designed and implemented; it provides ways to manage the collected data sets, browse and search their individual tweets, and provide overview visualizations on the data sets, such as providing a histogram that displays the number of tweets collected per day of the event. It addressed the limitations of EPIC Collect by providing a unified platform for any analyst to access the collected data in a straightforward manner. Furthermore, EPIC Analyze provides excellent query support that allows analysts to filter a large data set down to a more manageable size and then store that smaller set of tweets as its own individual data set, allowing analysts to return to it quickly.

However, EPIC Analyze is not without its own limitations. Since it relies on EPIC Collect as its data provider, data processing and indexing has to be performed as batch processes. This constraint meant that newly collected tweets for active events were not accessible within EPIC Analyze in real-time. Another limitation is that data sets stored in EPIC Collect are not migrated automatically to EPIC Analyze; as a result, analysts have to rely on developers to make a data set accessible. Lastly, a final limitation is that EPIC Analyze is only accessible as a web-based application; it does not provide any APIs so that analysts with programming knowledge have easy access to raw data.

IDCAP is a redesign of EPIC Collect to introduce incremental indexing capabilities that go beyond just reliably storing Twitter data in a NoSQL database. Its data model stores Twitter data in a way that provides answers to common queries in constant time. IDCAP's design almost eliminates the need to batch process collected data sets since every incoming tweet is being incrementally processed and indexed as it arrives. Of course, data sets that were collected before IDCAP was created still need to be batch processed to be migrated into IDCAP, and there are a variety of queries that are not directly supported by IDCAP's data model. Such queries have to be answered by iterating through all tweets for a particular data set, which is a form of batch processing. However, for those queries directly supported by IDCAP's data model (and the designer of IDCAP made every attempt to support the most common queries that analysts make), no additional work is needed after a tweet has been stored. The answers to those queries are updated as each new tweet is added to a data set. The limitations that IDCAP has, then, is the fact that it must fall back to batch processing for queries not directly supported by its data model. Furthermore, it makes no attempt to support full-text search over its collected data sets. I address these limitations in my own work, discussed below.

EPIC Real-Time is a microservices-based big data analytics platform discussed above. Despite the fact that the platform was based on a user study to determine what analysts needed from a data-intensive software system for crisis informatics research, it is not free from limitations. Similar to IDCAP, its data model has no support for a full-text search on its collected data. Another limitation is that designing a new type of query for the system requires a great deal of skill and knowledge of the Lambda architecture, making it difficult for analysts to extend it.

For my research, I investigated the types of software architecture and software infrastructure needed in data-intensive software systems to ensure that all available analysis techniques for a system have been applied to all available data sets collected for that system and that the progress of performing this task is visible to the analysts that make use of that system.

To support this task, my design includes a monitoring system to keep track of the progress of each analysis technique on any given data set. For instance, as an analysis technique processes individual posts (e.g. tweets) within a data set, the monitoring system updates its database with that information. If my software infrastructure encountered a problem and had to be restarted, the monitoring system's database would allow my system to pick up its analysis where it left off, avoiding the duplication of work that would occur if the same technique were asked to process a post it had seen before.

Furthermore, my design allows my infrastructure to apply analysis techniques both to historical data sets and to data sets under active collection in a uniform manner. This feature allows my system to provide analysts with an up-to-date view of all their data sets no matter when they were collected. A final feature allows my system to be extensible with respect to analysis techniques and with respect to new data sets, both of which can be added at any time. If analysis techniques are added, then all previously processed data sets are updated by having the new analysis technique applied to them. If a new data set is added, then all analysis techniques are applied to it successfully until all queries and analysis results have been computed for it. These features address the limitations discussed above of the current set of Project EPIC software.

Chapter 4

Research Questions

With the previous sections serving as context, I can now present my research questions.

RQ1: What software architecture will support the uniform processing of historical and streaming data to allow the automation of applying analysis techniques to that data efficiently?

RQ2: What techniques are needed to ensure that an extensible set of analysis techniques can be applied to newly collected data to minimize the use of batch processing within the system?

RQ3: What data model and what techniques are needed to efficiently track the status of applying an analysis technique to a large data set? How does one enable the granular tracking of the processing of individual posts within a data set?

Chapter 5

System Design and Implementation

To explore my research questions, I enhanced the existing Project EPIC infrastructure with new capabilities. I built on the work of IDCAP by reusing its data model to archive collected tweets and I provided a mechanism by which analysis techniques become first-class entities in the infrastructure; my infrastructure allows analysis techniques to be added, removed, and/or changed as needed by analysts.

In my prototype, I implemented a suite of layers and services that collectively provide an execution environment for analysis techniques, ensuring that each analysis technique is applied to each data set until all data sets have been processed. The services also ensure that tweets are collected/archived and monitors the progress of the analysis techniques so that analysts can understand what work has been performed and what remains to be done.

This approach helps to address several concerns that Project EPIC analysts had with prior versions of the Project EPIC infrastructure. The batch processing approach of EPIC Analyze meant that analysts could not use it to monitor events under active collection. Some events are collected for months; this limitation was hence a major roadblock with respect to adoption of EPIC Analyze. The ability to uniformly apply analysis techniques to both historical data sets and data sets under active collection addresses this significant limitation.

Treating analysis techniques as first-class entities helps address a set of limitations of EPIC Analyze that again lowers the barrier for adoption. Having first-class analysis techniques allow for my prototype to provide clear instructions to analysts on how to "wrap" a technique they have been working on into a unit of code that can be plugged into my system. Once they have performed the work of creating an analysis technique, they can be assured that it will eventually be applied to all data sets, eliminating the need for them to apply it manually to only a subset of data sets. This mechanism also provides extensibility allowing new techniques to be developed and plugged in efficiently. Finally, the monitoring subsystem provides much needed reliability and visibility into a process that is otherwise manual and error prone.

I now present the design of my prototype; as mentioned earlier, I use the IDCAP data model to archive twitter data; this data is also stored in the queues created for each event (as discussed below). My design is also inspired by elements of the Lambda and Kappa architectures to help me design real time incremental processing, batch processing, data queuing, monitoring, and recovery functionality into my system in a reliable, efficient, and scalable way. My design is also based on the microservices paradigm such that each service in the prototype can be added or removed without causing a total halt of the entire infrastructure.

5.1 Architecture

My system is composed of four main layers: the queuing layer, the services layer, the data processing layer, and the storage layer (see Fig. 5.1). I describe these layers in more detail below.

5.2 Queuing Layer

The queueing layer is responsible for transporting individual elements of a data set around the system in support of the processing, monitoring, storage, and recovery functions. It also serves as the means for data from external sources to enter the system in the first place. Finally, it aids in system reliability and scalability as modern queueing software can be run on multiple machines to provide high availability of *virtual queues* and offer features such as the ability to create *durable queues* that persist their content on disk to allow them to be recovered after a system failure.

For my dissertation, I designed a set of queues that are used to support the task of automating the application of analysis techniques to large data sets. In my prototype, I have a queue connected

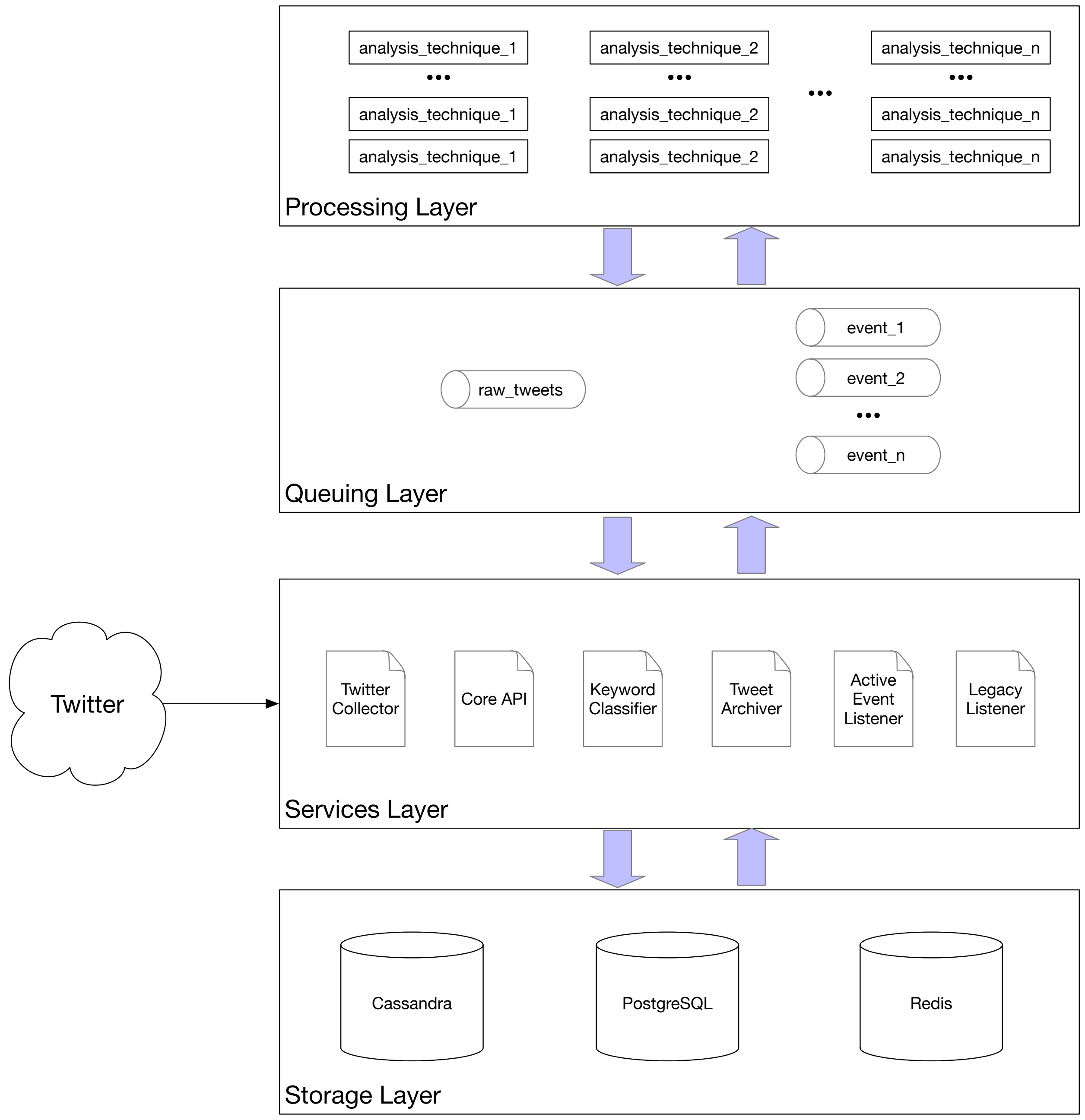


Figure 5.1: The Software Architecture

to the Twitter Streaming API that will feed unprocessed Twitter JSON objects into a `raw_tweets` queue that will then be the primary source of input to all of the other subsystems. In particular, each tweet in the input queue will need to be analyzed to determine why it was collected (there can be multiple active events each with their own set of keywords). Once a tweet has been classified, a copy of it is placed in a queue corresponding to each of the events to which it belongs. The event queue then feeds tweets to multiple queues each tied to a different analysis technique. The software needed to perform that analysis technique will consume tweets from that queue, perform its analysis, and then update databases as needed, including the database maintained by the monitoring system.

5.2.1 Kafka

Apache Kafka, is a modern scalable distributed queuing system used for real time streaming of high throughput data [64, 32]. To manage and process large amounts of user activity data generated by hundreds of millions of LinkedIn users, and to supply the high volume of generated data to integrate with the company's systems and products, LinkedIn developers built a pub/sub message logging system as their "centralized event pipelining platform." This platform, Kafka, was distributed, scalable, and handled high data volumes [64].

In my prototype, I rely on Kafka for my queues. I chose Kafka because it supports the option of retaining messages forever [64, 17]. In addition, according to Jay Kreps, the creator of the Kappa Architecture, there are some compelling use cases where Kafka should be used to store data for long term purposes, and one of those use cases is re-processing data [35].

In my system, I need to run multiple processing and analysis methods on Project EPIC's collected data sets. Different analysts might have different methodologies for processing the same event's tweets to find interesting insights about the collected data. Therefore, using Kafka topics for data retention suits our needs for executing various analysis procedures on our data whether its associated event is active or historical. Retaining messages infinitely in Kafka is done by changing the data retention setting for Kafka queues to "-1". When a new event's queue is created, we provide this setting as `ms.retention=-1`. The data retention setting for the `raw_tweets` queue is not configured to persist forever. It is instead set to twenty-four hours. The reason for making this choice is that we do not need to keep unclassified tweets forever because there is no need to re-run analysis on them, and we want to avoid data bloat in that queue. Therefore, we only keep raw unclassified tweets for the defined retention time after which Kafka will discard them automatically.

5.2.2 Issues with an alternate approach

Before I landed on choosing Apache Kafka as my data store, I tried a different approach in which we did not make use of Kafka to retain tweets in each event's queue. The original approach was that once a tweet was classified, it would be sent to its designated event queue and then stored in Cassandra. An event's queue was assigned a short retention time to reduce a storage size and thus I only relied on Cassandra to store tweets. When an event's tweets needed to be processed by an analysis technique, my system would instantiate a new queue dedicated to feed an event's data to that analysis technique by reading data from Cassandra. The approach was to use Cassandra for two reasons: 1) to permanently store an event's tweets and 2) and to use it as the way to provide data to an analysis technique. This approach is documented in Fig. 5.2.

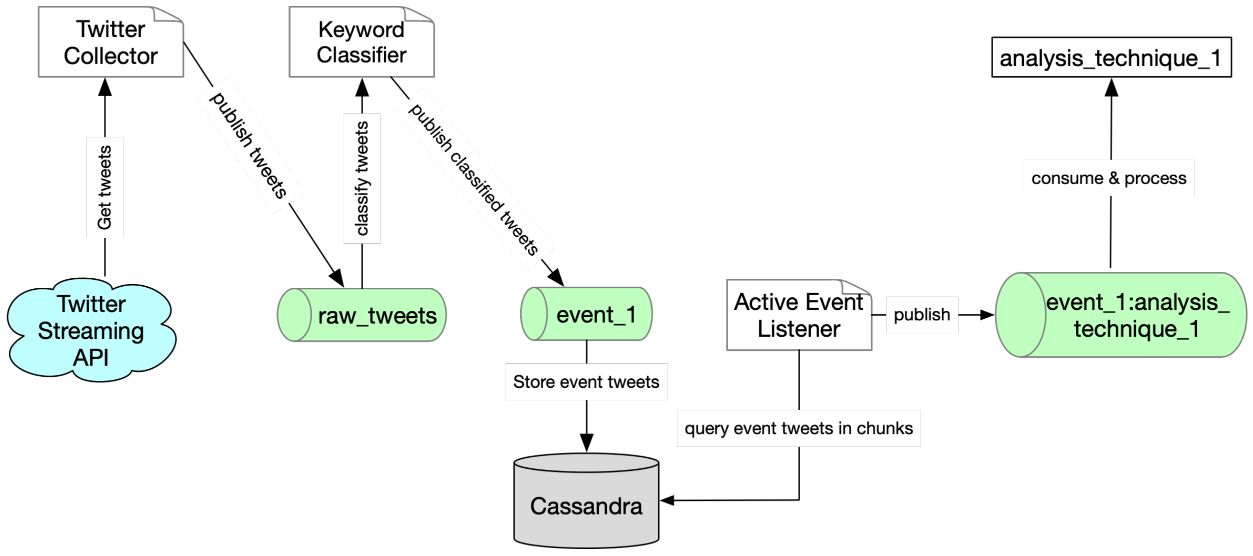


Figure 5.2: The process of the old approach

That strategy worked well for historical events since those events were not actively collecting data. It could easily read all of the tweets from an historical event and push them into the queues related to each analysis technique. For example, if an event had one hundred thousand tweets, an analysis technique would process the exact amount by having a counter that increments when a tweet was processed; thus, that counter should match with a historical event's total tweets. I tested that to process a historical event called "2012 Casa Grande Explosion," which has 183,738 tweets with an analysis technique that provides histogram of the most popular hashtags called "Top Hashtags." Every time I processed that event, the total tweet processing counter always matched the event's total tweet count.

When I ran tests with a simulated active event and an actual twitter collection event, I noticed issues related to inconsistencies with the total amount of tweets for that event and the total amount of tweets processed. For an active event, it makes sense that total processed tweets would be smaller than the actual amount of tweets collected in an event. However, the processing counter should eventually equal the total collected tweets once an event is deactivated. I discovered that once an event was deactivated, the total processed tweets would not match the total number of tweets of an event. I first simulated an active event by feeding the `raw_tweets` queue with JSON data from the "2012 Casa Grande Explosion" event stored separately in a JSON file. The event was deactivated when it reached its original total of 183,738 which also corresponded to the end of the JSON file. Once deactivated, I found that the processing also finished its job but processed only 161,689 tweets, which means the system was dropping 12% of the total unprocessed tweets. I reran the simulation and found that only 113,917 tweets were processed, meaning that the system had dropped 38% of the total tweets. That was a drastic change. With each rerun, the architecture shown in Fig. 5.2 had different random results and never actually successfully completed the processing of the simulated event.

I then created a sample event that collected tweets with high frequency terms, such as the smiley face emoji. I ran a collection for five minutes for the first time which collected 12,713 tweets. Once deactivated, I found that the processing also finished its job but processed only 10,481 tweets (dropping 17% of the total number of tweets). Since this event was now a historical event, I reran the processing and the total amount of processed tweets matched with the total data for that event. This eliminated any doubts I had that it was the analysis technique that was dropping the tweets.

After noticing these inconsistent results of unprocessed tweets, I compared the amount of tweets in the event's queue with the analysis technique's queue for this event. For both simulation and actual twitter collection tests, I discovered that the analysis technique's queue always had fewer tweets than the actual total tweets that existed in the event's queue. I thought that my code had bugs and was not properly publishing tweets to the analysis technique's queue for both events. After further tests, I realized that it was properly publishing tweets for historical events but not for the active ones that were collecting tweets in real-time. After further investigation, the outcome was that Cassandra was not a reliable option for streaming real time data.

Cassandra uses CQL (Cassandra Query Language) which is similar to SQL; it is used to simplify querying Casandra for people familiar with SQL. Cassandra is known to have fast writes [39]. One of the challenges we realised was reading newly inserted data from Cassandra. In the case of an active event, new streams of data are inserted into the database which is not an issue. The problem is reading those new data. When reading from Cassandra, the CQL API allows paginating large queries into manageable chunks to not exhaust a machine's resources. However, it allows only reading the amount of data for the time when the query was requested. For example, if 1000 records were stored by the time the query was issued, then we need to run another query to read any additional records added after the first query, and the API does not allow to read from the last point that was read in the column family. So I modified the data model and added time stamps when inserting tweets so that I can query from the last time stamp. I also modified my script to wait for a few seconds to give some time to buffer results before querying. I still encountered the problems described above.

Despite the fact that reads are slower than writes in Cassandra, high volume data writes are handled asynchronously for replication [39]. As a result, each time my system was trying to read data stored for an active event, there were tweets that had not yet been written to the column family and Cassandra's API had no way to reveal that situation to my software and so its queries for the most recent set of tweets were always missing tweets that had not yet been written. Thus, querying reads in small chunks and providing a wait buffer did not solve the problem.

5.2.3 The new approach

The inconsistency problem led me to use Cassandra only to archive tweets using IDCAP's data model. I had to think differently about conserving storage space. I instead decided to use Kafka to retain an event's data in each event's dedicated queue and use more storage space across the entire system since storage is (relatively) cheap. This decision eliminated the need for dedicated queues for each analysis technique. Since each event's queue now stores all tweets ever collected for that event, adding another dedicated queue for an analysis technique for that event is redundant. I instead settled on an approach whereby an analysis technique subscribes to an event's queue and processes the data directly as it becomes available.

5.3 The Services Layer (Core Services)

The services layer of my architecture is home to a set of **core services**. Each core service is implemented as a microservice and is responsible for providing a key feature of my prototype. These services collect tweets, archive them, and send them to analysis techniques for processing. These services can scale both vertically and horizontally on the cloud as dictated by resource demands. My prototype has the following six core services:

(1) The Core API

(2) Active Event Listener

(3) Legacy Listener

(4) Twitter Collector

(5) Keyword Classifier

(6) Tweet Archiver

The following sections will describe and explain each of these services in more detail.

5.3.1 The Core API

This service implements a RESTful API that provides functionality for managing the primary concepts associated with my system: Events, Processing Techniques, and Jobs.[1] The metadata associated with these concepts are stored in database tables in PostgreSQL and the Core API is responsible for creating, editing, and deleting instances of these concepts in response to calls from users and other services in the prototype. A user of my prototype can only create or edit Events and Processing Techniques; Jobs, on the other hand, are only created in response to requests from the Active Event and Legacy listeners. I will now describe each of these concepts in more detail.

5.3.1.1 Events

An event is a concept in my system that is related to a real-world mass convergence event, such as a natural disaster or crisis event, involving members of the public. When an analyst wants to study such an event, they create an instance of an event in my prototype and associate keywords with the event that will be used to search for tweets via Twitter's Streaming API. In my prototype, a database table is created in PostgreSQL to keep track of the metadata for each event. (See Table 5.1.)

When a new event is created, the system will create a Kafka queue based on that event's name. For example, if one creates an event called "2019 Hurricane Dorian," the queue will be created as `2019_hurricane_dorian`. The Core API will also send a message to a Pub/Sub channel in Redis called `new_event_created`. That message is a JSON representation of the new event. I will describe how my prototype makes use of Redis (an in-memory database) in more detail below.

When an event is deactivated by setting its "active" field to false, an updated JSON representation of the event sent to the Pub/Sub channel in Redis called `existing_event_deactivated`.

[1] Note: I use "analysis technique" and "processing technique" interchangeably in this document.

FIELD	DESCRIPTION
`name`	the name of an event; these names usually start with a year followed by descriptive text (e.g. "2019 Hurricane Dorian")
`terms`	a list of keywords relevant for an event, used for collecting tweets
`active`	a Boolean value that indicates whether an event is under active collection. The default value is true when an event is first created
`qname`	an event's queue name; this name is auto-generated by the system and is used in Kafka for creating queues related to the event (e.g. `2019_hurricane_dorian`)
`created_at`	an event's creation date
`updated_at`	an event's modification date; note: the only times an event is modified is when it is being deactivated or its list of keywords is being updated.

Table 5.1: The database table for an Event

Additionally, the Core API publishes a JSON message to the Kafka queue of the now deactivated event to indicate that the event is no longer active. The message sent is `{"status": "closed"}`.

Otherwise, an event can be updated by providing a new list of keywords. When this happens, the Core API sends an updated JSON representation of the event to the Pub/Sub channel in Redis called `existing_event_changed`. This channel is used by the Twitter Collector to know when to restart its connection to Twitter to start collecting with the updated set of keywords.

I now provide a description of all endpoints for the Event's REST API. These endpoints allow other services to manage events.

Create an event:

- *URL:* [POST] `/events`
- *Request Body:*

```
{"name": "EVENT_NAME", "terms": ["term 1", "term 2", ..., "term n"]}
```

- *Example Response:*

```
{
  "id": 1,
  "name": "2019 United Nations General Assembly",
  "terms": [
    "UNGA",
    "#UNGA",
    "#UNGA2019",
    "greta thunberg"
  ],
  "active": false,
  "created_at": "2019-09-24T21:41:46.409Z",
  "updated_at": "2019-09-24T21:43:20.122Z",
  "qname": "2019_united_nations_general_assembly"
}
```

Get all events:

- *URL:* [GET] `/events`
- *Response:*

```
[
  {
    "id": 2,
    "name": "2019 Republicans",
    "terms": [
      "gop",
      "trump",
      "republican",
      "republicans"
    ],
    "active": false,
    "created_at": "2019-09-24T21:42:27.441Z",
    "updated_at": "2019-09-24T21:43:06.321Z",
    "qname": "2019_republicans"
  },
  {
    "id": 1,
    "name": "2019 United Nations General Assembly",
    "terms": [
      "UNGA",
      "#UNGA",
      "#UNGA2019",
      "greta thunberg"
    ],
    "active": false,
    "ended_at": null,
    "created_at": "2019-09-24T21:41:46.409Z",
    "updated_at": "2019-09-24T21:43:20.122Z",
    "qname": "2019_united_nations_general_assembly"
  }
]
```

Get a specific event:

- *URL:* [GET] `/events/:id`
- *Response:*

```
{
 "id": 1,
 "name": "2019 United Nations General Assembly",
 "terms": [
   "UNGA",
   "#UNGA",
   "#UNGA2019",
   "greta thunberg"
 ],
 "active": false,
    "created_at": "2019-09-24T21:41:46.409Z",
    "updated_at": "2019-09-24T21:43:20.122Z",
    "qname": "2019_united_nations_general_assembly"
  }
```

Get active or historical events:

- *URL:* [GET] `/events/active`
- *Request Body:*

  ```
  {"active": true}
  ```

- *Response:*

  ```
  [
   {
     "id": 2,
     "name": "2019 Republicans",
     "terms": [
       "gop",
       "trump",
       "republican",
       "republicans"
     ],
     "qname": "2019_republicans"
   },
   {
     "id": 1,
     "name": "2019 United Nations General Assembly",
     "terms": [
       "UNGA",
       "#UNGA",
       "#UNGA2019",
       "greta thunberg"
  ```

```
    ],
    "qname": "2019_united_nations_general_assembly"
  }
]
```

Search an event by name:

- *URL:* [GET] `/events/search`
- *Request Body:*

```
    {"name": "2019 Hurricane Dorian"}
```

- *Response:*

```
 {
"id": 1,
"name":"2019 Hurricane Dorian",
"terms":["hurricane dorian","#hurricaneDorian", "#dorian"]},
"active": true,
"created_at": "2019-09-24T21:41:46.409Z",
"updated_at": "2019-09-24T21:43:20.122Z",
"qname": "2019_hurricane_dorian"
}
```

Edit a specific event:

- *URL:* [PUT] `/events/:id`
- *Request Body:*

```
    {"active": false}
```

- *Response:*

```
 {
"id": 1,
"name":"2019 Hurricane Dorian",
"terms":["hurricane dorian","#hurricaneDorian", "#dorian"]},
"active": false,
"created_at": "2019-09-24T21:41:46.409Z",
"updated_at": "2019-09-24T21:43:20.122Z",
"qname": "2019_hurricane_dorian"
}
```

5.3.1.2 Processing Techniques

A processing technique is the concept used to integrate an analysis technique into the prototype. The table captures the name of the technique and the URL of its API. It also keeps track of when the technique was added to the system and whether it is still active and should be applied to all events. If it has been retired, it also tracks when the technique was removed from the system. (See Table 5.2.)

FIELD	**DESCRIPTION**
`name`	a descriptive name of a processing technique (e.g. “Popular Hashtags”)
`url`	the URL of a processing technique’s API (e.g. `http://example.com:4001/top_hashtags`)
`active`	a Boolean field that indicates whether a technique is available for processing events; a value of `false` indicates that the processing technique has been retired and will no longer be used
`created_at`	the date/time when a processing technique was added to the system
`updated_at`	the date/time when a processing technique was retired from the system

Table 5.2: The database table for a Processing Technique

When a new processing technique is added, the Core API sends a message to the Pub/Sub channel in Redis named `new_processing_technique_created`. That message is a JSON representation of the new processing technique.

I now provide a description of all endpoints for the Processing Technique’s REST API. These endpoints allow other services to manage processing techniques.

Create a processing technique:

- *URL:* [POST] `/processing_techniques`
- *Request Body:*

```
    {"name": "Top Languages",
  "url": "http://192.168.99.100:4003/top_languages"}
```

- *Response:*

```
{
  "id": 4,
  "name": "Top Languages",
  "url": "http://192.168.99.100:4003/top_languages",
  "active": true,
  "created_at": "2019-09-30T20:23:35.544Z",
  "updated_at": "2019-09-30T20:23:35.544Z"
}
```

Get all processing techniques:

- *URL:* [GET] `/processing_techniques`
- *Response:*

```
[
  {
    "id": 1,
    "name": "Top Hashtags",
    "url": "http://192.168.99.100:4001/top_hashtags",
    "active": true,
    "created_at": "2019-09-11T05:02:25.907Z",
    "updated_at": "2019-09-11T05:02:25.907Z"
  },
  {
    "id": 2,
    "name": "Top Users",
    "url": "http://192.168.99.100:4002/top_users",
    "active": true,
    "created_at": "2019-09-11T05:06:02.290Z",
    "updated_at": "2019-09-11T05:06:02.290Z"
  }
]
```

Get a specific processing technique:

- *URL:* [GET] `/processing_techniques/:id`
- *Response:*

```
 {
  "id": 1,
  "name": "Top Hashtags",
  "url": "http://192.168.99.100:4001/top_hashtags",
  "active": true,
  "created_at": "2019-09-24T21:40:25.122Z",
  "updated_at": "2019-09-24T21:40:25.122Z"
}
```

Search a processing technique by name:

- *URL:* [GET] `/processing_techniques/search`
- *Request Body:*

```
{"name": "Top Hashtags"}
```

- *Response:*

```
 {
  "id": 1,
  "name": "Top Hashtags",
  "url": "http://localhost:4001/top_hashtags",
  "active": true,
  "created_at": "2019-04-12T00:21:41.727Z",
  "updated_at": "2019-04-12T00:21:41.727Z"
}
```

Edit a specific processing technique:

- *URL:* [PUT] `/processing_techniques/:id`
- *Request Body:*

  ```
  {"active": false}
  ```

- *Response:*

```
{
  "id": 1,
  "name": "Top Hashtags",
  "url": "http://localhost:4001/top_hashtags",
  "active": false,
  "created_at": "2019-04-12T00:21:41.727Z",
  "updated_at": "2019-04-12T00:23:05.562Z"
}
```

5.3.1.3 Jobs

A job is a concept for tracking when an analysis technique is actively processing tweets for a given event. The database table associated with jobs is responsible for keeping a history of which events have been processed by which processing techniques and what the status is of each job. This particular database table is a join table that uses an Event's id and a Processing Technique's id as foreign keys. It is de-normalized so it has accessible information about an event and a processing technique to significantly reduce the number of JOIN queries that need to be made on the database.

By de-normalized, I mean that information from the Events and Processing Techniques tables are duplicated in the Jobs table, such as an event's name or the URL of a processing technique's API. The specific fields are listed in Table 5.3.

FIELD	**DESCRIPTION**
`event_id`	an event's unique id from the Events table
`processing_technique_id`	a processing technique's unique id from the Processing Techniques table
`event_name`	the events name that is being processed
`pt_name`	the processing technique's name that is processing an event
`pt_url`	the location of a processing technique's API
`status`	a job's status. One of: `pending`, `running`, or `finished`
`created_at`	a job's creation date
`updated_at`	a job's completion date

Table 5.3: The database table for a Job

I now provide a description of all endpoints for the Job's REST API. These endpoints allow other services to manage jobs.

Create a Job:

- *URL:* [POST] `/jobs`
- *Request Body:*

```
{"event_id": 2,
"processing_technique_id": 1,
"event_name": "2019 Republicans",
"pt_name": "Top Hashtags",
"pt_url": "http://192.168.99.100:4001/top_hashtags",
"status": "pending"}
```

- *Response:*

```
{
  "id": 1,
  "event_id": 2,
  "processing_technique_id": 1,
  "event_name": "2019 Republicans",
  "pt_name": "Top Hashtags",
  "pt_url": "http://192.168.99.100:4001/top_hashtags",
  "status": "pending",
  "created_at": "2019-09-30T20:17:36.403Z",
  "updated_at": "2019-09-30T20:17:40.647Z"
}
```

Get all jobs:

- *URL:* [GET] `/jobs`
- *Response:*

```
[
  {
    "id": 1,
    "event_id": 2,
    "processing_technique_id": 1,
    "event_name": "2019 Republicans",
    "pt_name": "Top Hashtags",
    "pt_url": "http://192.168.99.100:4001/top_hashtags",
    "status": "finished",
    "created_at": "2019-09-30T20:17:36.403Z",
    "updated_at": "2019-09-30T20:17:40.647Z"
  },
  {
    "id": 2,
    "event_id": 2,
    "processing_technique_id": 2,
    "event_name": "2019 Republicans",
    "pt_name": "Top Users",
    "pt_url": "http://192.168.99.100:4002/top_users",
    "status": "finished",
    "created_at": "2019-09-30T20:17:36.537Z",
    "updated_at": "2019-09-30T20:17:40.824Z"
  },
  {
    "id": 4,
    "event_id": 1,
    "processing_technique_id": 1,
    "event_name": "2019 United Nations General Assembly",
    "pt_name": "Top Hashtags",
    "pt_url": "http://192.168.99.100:4001/top_hashtags",
    "status": "finished",
    "created_at": "2019-09-30T20:17:38.070Z",
    "updated_at": "2019-09-30T20:17:42.410Z"
  },
  {
    "id": 5,
    "event_id": 1,
    "processing_technique_id": 2,
    "event_name": "2019 United Nations General Assembly",
    "pt_name": "Top Users",
    "pt_url": "http://192.168.99.100:4002/top_users",
    "status": "finished",
    "created_at": "2019-09-30T20:17:38.256Z",
    "updated_at": "2019-09-30T20:17:42.596Z"
  }
]
```

Get a specific job:

- *URL:* [GET] `/job/:id`

- *Response:*

```
{
  "id": 4,
  "event_id": 1,
  "processing_technique_id": 1,
  "event_name": "2019 United Nations General Assembly",
  "pt_name": "Top Hashtags",
  "pt_url": "http://192.168.99.100:4001/top_hashtags",
  "status": "finished",
  "created_at": "2019-09-30T20:17:38.070Z",
  "updated_at": "2019-09-30T20:17:42.410Z"
 }
```

Search a job by an event's name, a processing technique's name, or both:

- *URL:* [GET] `/jobs/search`

- *Request Body:*

```
  {"event_name": "2019 United Nations General Assembly", "pt_name": "Top
     Hashtags"}
```

- *Response:*

```
[
 {
  "id": 4,
  "event_id": 1,
  "processing_technique_id": 1,
  "event_name": "2019 United Nations General Assembly",
  "pt_name": "Top Hashtags",
  "pt_url": "http://192.168.99.100:4001/top_hashtags",
  "status": "finished",
  "created_at": "2019-09-30T20:17:38.070Z",
  "updated_at": "2019-09-30T20:17:42.410Z"
  }
]
```

Edit a specific job:

- *URL:* [PUT] `/jobs/:id`
- *Request Body:*

```
{"status": "finished"}
```

- *Response:*

```
{
   "id": 4,
   "event_id": 1,
   "processing_technique_id": 1,
   "event_name": "2019 United Nations General Assembly",
   "pt_name": "Top Hashtags",
   "pt_url": "http://192.168.99.100:4001/top_hashtags",
   "status": "finished",
   "created_at": "2019-09-30T20:17:38.070Z",
   "updated_at": "2019-09-30T20:17:42.410Z"
}
```

5.3.2 Active Event Listener

The Active Event Listener is a service that creates new jobs after an event or processing technique is created. It listens to the `new_event_created` and `new_processing_technique_created` Pub/Sub channels in Redis and performs the appropriate actions in response to any new messages on those channels. If a new event is created, then the listener retrieves all of the active processing techniques and creates a new job for each one for the new event. If a new processing technique is created, the listener retrieves all of the active events and creates a new job for each one for the new technique. In this way, the system ensures that analysis is being performed on all active events.

5.3.3 Legacy Listener

The Legacy Listener is almost identical to the Active Event Listener. The difference is that it creates new jobs specifically for historical events that are no longer active. The Legacy Listener listens to the `new_processing_technique_created` Pub/Sub channel in Redis and performs the appropriate action in response to any new messages on that channel. If a new processing technique is created, the listener retrieves all of the inactive events and creates a new job for each one for the new technique. In this way, the system ensures that analysis is also being performed on all inactive events when a new processing technique is added to the system.

5.3.4 Twitter Collector

The Twitter Collector is the service that connects to Twitter's Streaming API and collects tweets for all active events. When this service starts, it uses the Core API to retrieve the keywords for all active events. It submits those keywords to the Streaming API and then processes all of the incoming data that Twitter provides. Some of the messages that come from Twitter are related to the service itself—compliance messages, rate limiting notifications, server side error messages, etc.—and are handled by sending them to a log to be archived for later review and analysis. The remaining messages are tweet objects in JSON format. These messages are added to the `raw_tweets` queue in Kafka.

The Twitter Collector also subscribes to the following Redis Pub/Sub channels: `new_event_created`, `existing_event_deactivated`, and `existing_event_changed`. When any of these channels publishes a message, they add those messages to the Twitter Collector's internal queue. Every ten seconds, the Twitter Collector checks its queue for updates. If the queue has messages, the collector stops the current twitter collection, fetches an updated list of keywords, and then restarts the collection by sending the updated keywords to Twitters Streaming API. The overhead of reconnecting to Twitter ranges between 0.5 seconds and 1 second. The reason to have a 10-second delay on checking the internal queue is to avoid disconnecting and reconnecting to

Twitters API every time there is a new event created, deactivated, or updated. Sending too many connection requests will cause Twitter to potentially throttle or rate-limit our connection.

5.3.5 Keyword Classifier

The Keyword Classifier is a service that given a tweet determines which active events were responsible for having collected. A single tweet can belong to multiple events if those events share a common keyword. For example, two events related to hurricanes can each contain the keyword `hurricane` and have tweets containing that word sent to both events.

When this service starts for the first time, it checks for existing active events by calling the Core API. It then uses their keywords to classify tweets. If the system starts and there are no active events, the service listens to the following Redis channels: `new_event_created`, `existing_event_deactivated`, and `existing_event_changed`. When any of these channels publish a message, the classifier updates its set of active keywords and restarts its classification process.

The classification is performed by consuming tweets from Kafka's `raw_tweets` queue. The classifier then compares a tweet's text fields with an event's keywords and sends the tweet to the queues of any matching events. The fields used to perform classification are a tweet's text, a retweet's text, a quoted tweet's text, URLs, user mentions, and hashtags. The classifier also checks for the "full text" attribute in a tweet, retweet, and quoted tweet. This attribute contains a tweet's text when that text is greater than Twitter's traditional limit of 140 characters.

As mentioned above, when a tweet matches with at least one of an event's keywords, it is published to that event's queue in Kafka. The published message is in JSON format and contains attributes for an event's name, a list of matched keywords, and the full tweet object:

```
{ "event": "EVENT NAME",
  "matched_kw": ["MATCHED", "KEYWORDS"],
  "tweet": {TWEET JSON} }
```

5.3.6 Tweet Archiver

The Tweet Archiver is a service that ensures that all tweets collected by my prototype are stored in Cassandra to archive them. I make use of the data model developed for IDCAP [5] to gain the benefits it provides, as described in Chapter 2.5.3. It does this by subscribing to each event queue in Kafka and ensuring that all tweets stored for an event get archived in Cassandra. When the Tweet Archiver starts for the first time, it checks for existing active events by calling the Core API. It then subscribes to the appropriate queues and starts archiving tweets for each event. If the system starts and there are no active events, the service listens to the `new_event_created` Redis channel. When that channel publishes a message, the Tweet Archiver will subscribe to its Kafka queue and will start archiving its tweets as they arrive. The archiver stops storing data for an event when an event is deactivated and receives the `{"status": "closed"}` message that is placed there by the Core API.

As mentioned in [5], each rowkey of an event in Cassandra stores 20,000 tweets except for the last rowkey, which will always be less than that threshold. Each rowkey for an event has an index, and that index will be incremented once that threshold is reached. For example, if we have an event called `EVENT_X`, the first rowkey will have an index of 1 and is represented as `event_x:1`. Once this rowkey reaches the twenty-thousand-tweets-per-rowkey limit, another rowkey is created with an index of 2 such as `event_x:2`. The Tweet Archiver continues to store tweets for an event in this fashion until an event is deactivated; it then stores any remaining tweets in the last rowkey. This mechanism is discussed in more detail below in Chapter 5.5.1.

5.4 Processing Layer

The Processing Layer consist of the technologies needed to analyze the collected data sets. This includes both the technologies needed to create and operate my prototype but also the software needed to implement each analysis technique and to update the storage layer once processing is complete. Data is consumed through the queues in the Queuing Layer; each analysis technique will independently process data and then store results in the appropriate database in the storage layer. I implement all of the analysis techniques as microservices to provide the ability of managing and scaling each feature independently and to add new features without affecting the rest of the system. A generic component is then configured to pull data off of an event queue and pass it to the analysis technique for processing.

5.4.1 Analysis Techniques

Analysis techniques are RESTful applications that consume data, process that data, and generate results that provide insights into an event for analysts. For Project EPIC, crisis informatics analysts care about studying social media data, specially Twitter data. The work in [29] conducted a study that identified the various types of data analysis that Project EPIC analysts need to investigate their collected data sets.

In this project, I reused some of those ideas for data analysis and recreated them such that they are analysis techniques that can be plugged into my system. In particular, I implemented them as containerized independent microservices that are deployable on the cloud. I have implemented four processing techniques to analyze our collected tweets for both active and historical events. These techniques are serve as proofs of concept that my prototype can be used to analyze real crisis events. These four techniques are:

(1) **Top Hashtags**: This technique identifies the five most popular hashtags for an event.

(2) **Top Users**: This technique identifies the top five most active users who tweeted and/or retweeted the most in an event.

(3) **Top Retweets**: This technique identifies the top five most retweeted tweets for a given event.

(4) **Top Languages**: This technique identifies the top five languages used in an event.

All these RESTful processing techniques were packaged as Docker containers and deployed using Kubernetes using Google Cloud. Since they are HTTP-based RESTful microservices, we expose each service's URLs and ports to allow our Core Services to use them, and we enabled the load balancer configuration in Kubernetes deployment files for each of the these analysis techniques to allows us to scale these techniques to handle heavy processing loads while maintaining a single point of access to them by my core services. The results of each processing technique for each event are stored in Redis in a channel that is named following the convention `processing_technique:event_name:result`.

These processing techniques were built to be simple without any advanced software frameworks or platforms like Apache Spark. However, since I use microservices, REST, and well-know data stores, I tried to build these analysis methods in a way that can be followed to add many different kinds of processing techniques and have the option to use any software framework as long as a developer follows some requirements to successfully integrate the new technique with my infrastructure.

5.4.2 The framework for building a processing technique

The goal of a processing technique is to process tweets for an event. The following items are the requirements for creating a processing technique that successfully integrates with my prototype:

- Uses the Core API to search for Events and update the status of Jobs
- Provide a working HTTP-based end point for its API
- Include a REST API with a POST request which accepts JSON data that includes the parameter event (e.g. `{"event": "event name"}`). This parameter tells the technique its associated event which allows it to call the Core API and search for events.
- Subscribe to an events Kafka queue to process its tweets
- Update the processing jobs status to running once it starts processing
- Update the processing jobs status to processed after the data processing is complete

- Provide a Redis key that is a counter for processed tweets. The naming convention for this key is `processing_technique_name:event_name:processed`
- Process both active and historical data

We will use “Top Languages” as an example to show how a processing technique can fulfill these requirements. First, for purposes of exposition, we assume that we are running on a local machine with the default address of “localhost”, we are using port 4003, and we provide an API endpoint called `top_languages`. Therefore, the full URL of the processing technique will be `http://localhost:4003/top_languages`. This API must implement an event handler for POST requests. Once my prototype calls this API and sends an event name, such as “2019 Hurricane Dorian” to it using `{event: 2019 Hurricane Dorian}`, the processing technique will now have access to the Core API to search for information related to that event. That is done by calling `http://localhost:3000/event/search` and passing the event name. Assuming that the event exists, the Core API will return with the event’s information which includes the event’s queue name. After that, the processing technique can start its data processing as a background process, and returns a response with a message saying that it started processing the event.

The background process first sets the Job’s status for the event to “running” by first searching the Core API’s `jobs/search` endpoint and passing a JSON body with the event’s name and the processing technique’s name like this:
`{"event_name": "2019 Hurricane Dorian", "pt_name": "Top Languages"}`. This call will return the job’s record which will then allow this service to correctly update that job’s status. This is done by calling a PUT request to `/jobs/[:id]` with the “id” that was received via the previous call. We then send the JSON body `{"status": "running"}` which will update the jobs status accordingly.

Once updated, the application starts consuming the event’s Kafka queue and processing the tweets. Whenever tweets are processed, the Redis key
`top_hashtags:2019_hurricane_dorian:processed` gets updated. The results of the processing

technique is stored in any desired data store. For this technique, I made use of Redis.

The job should stop processing when it reads the JSON message `{"status": "closed"}` which indicates that the event is deactivated. The final step should be to store the last processed tweets, update the jobs status to "finished" by first searching the Core API's `jobs/search` endpoint. This will return the job's record which will then allow this service to correctly update that job's status. This is done by calling a PUT request to `/jobs/[:id]` where the "id" was received earlier with the job search. We then use the JSON body `{"status": "finished"}` which will update the job's status.

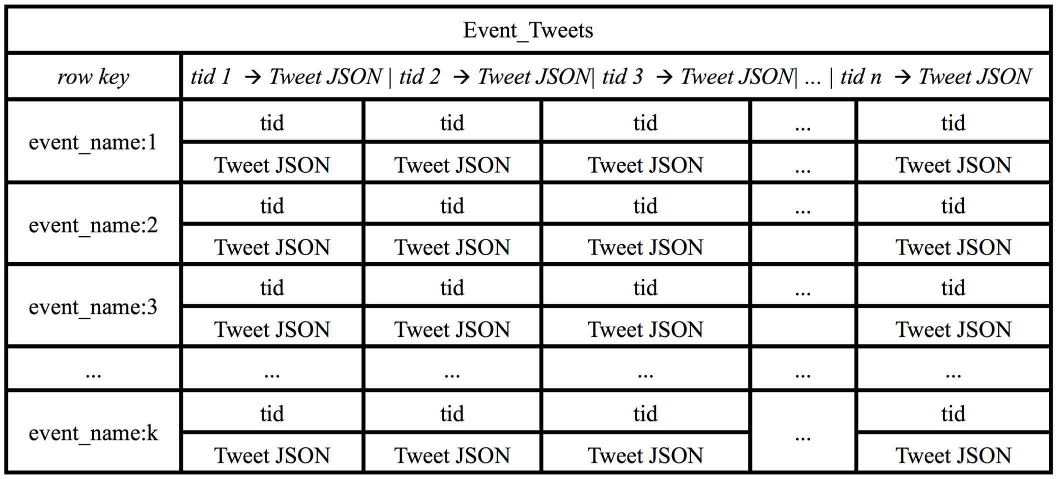
5.5 Storage Layer

The storage layer consists of multiple database technologies such as Cassandra, Redis, and PostgreSQL, to help store all of the data generated by the processing layer. Cassandra is responsible for archiving all tweets while the other databases are used to support tasks that best suit their design. Redis is used to store the output of processing techniques and the count of the tweets they have processed so far. It is also utilized as a message broker that provides Pub/Sub capabilities to coordinate the microservices in the Services Layer. PostgreSQL is used to store information for events, processing techniques, and jobs as discussed above.

5.5.1 Cassandra

Apache Cassandra is a NoSQL database built at Facebook to handle fast writes for immense amounts of incoming data while storing them on multiple nodes in a distributed fashion [39]. I use Cassandra for that reason and I use it to store tweets in their original JSON format. The data model used in this prototype is based on the work in [5]. Its data model already provides features for the efficient storage of Twitter data in Cassandra (see Fig. 5.3) and for building indexes on top of those tweets (see Fig. 5.4).

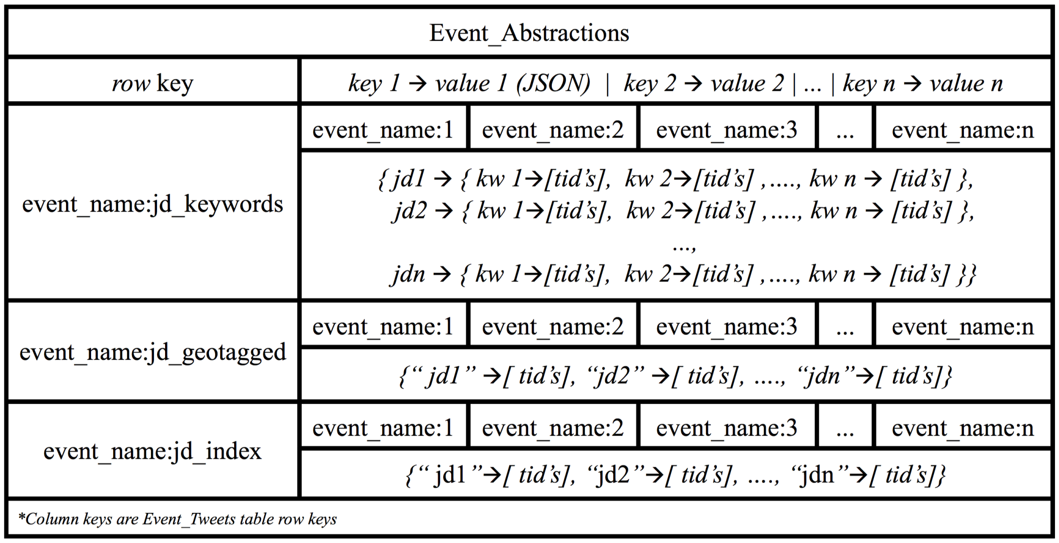
In the Services Layer, one of the core services that stores tweets to Cassandra is the Tweet Archiver described in 5.3.6. It follows the guidelines mentioned in [5] of storing tweets for an event



Event_Tweets					
row key	*tid 1 → Tweet JSON | tid 2 → Tweet JSON| tid 3 → Tweet JSON| ... | tid n → Tweet JSON*				
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 5.3: The IDCAP column family design for storing tweets efficiently [5].

in a row key where a defined threshold for each row is 20,000 tweets for the `Event_Tweets` column family shown in Fig. 5.3. Additionally, it incrementally stores the indices for each event in the `Event_Abstractions` column family, such as geotagged tweets for that event, the keywords used by date, and the tweets collected per day of an event as shown in Fig. 5.4.



Event_Abstractions

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 IDCAP column family design for building indexes on top of IDCAP's `Event_Tweets` column family [5]. This column family provides support for queries related to an event's keywords by date; its geotagged tweets; and all tweets collected on a given day. Additional indexes can be added by adding new row keys to this column family.

5.5.2 PostgreSQL

PostgreSQL [51] is a popular and open source relational database which is based on Stonebraker's and Rowe's work on POSTGRES [60]. We make use of this database to store the Events, Processing Techniques, and Jobs information that was presented in Tables 5.1, 5.2, and 5.3. The Core API relies on these tables and provide a JSON-based RESTful API to interact with them.

5.5.3 Redis

Redis [52] is a multipurpose, scalable, and high performance in-memory based open-source NoSQL data store. It is commonly used for caching, and it also functions as a data store and message broker.

In my prototype, I use the message broker feature of Redis as the means for making my event-driven microservices isolated and independent. By "event-driven," I simply mean that my microservices are designed to wait for messages to be published to various Redis channels. Once a message is received, they act as described in the sections above. All my Core Services use the event-driven paradigm which allows them to operate independently in an asynchronous fashion. The Core API triggers events by publishing messages to Redis channels, and the rest of the services subscribe to those channels and handle the incoming messages and execute the desired set of tasks. Table 5.4 describes the Redis channels and keys used in my prototype.

FIELD	**DESCRIPTION**
`new_event_created`	A channel that advertises when a new *Event* is created
`existing_event_deactivated`	A channel that advertises when an *Event* is deactivated
`existing_event_changed`	A channel that advertises when an *Event's* keywords have been updated
`new_processing_technique_created`	A channel that advertises when a new *Processing Technique* is created
`existing_processing_technique_changed`	A channel that advertises when a *Processing Technique* is updated
`processing_technique:event_name:processed`	A string used as a counter for the number of tweets processed for an event by a particular processing technique

Table 5.4: Redis channels and keys

5.6 Containerization

My prototype is based on the microservices architecture in which each service should be loosely coupled, scalable, and independently deployable. Packaging these microservices into virtualized container images allows each microservice to be deployed independently and scaled when it needs more resources. I use Docker [18], which is a container technology that enables applications with their dependencies to be packaged and deployed easily on cloud or on-premise container providers. This is similar to packaging Java applications into JAR files that can be used and imported into any JVM-based language. A container is an isolated unit which solves the “dependency hell” [8] problem that cause conflicts with other software dependencies which then might result in breaking other applications.

All core services and processing techniques are containarized microservices that are published to an online container hosting repository known as Docker Hub (`hub.docker.com`). As for data store technologies like Cassandra, Redis, and others, I use public container images offered by the creators of those technologies on Docker Hub, and I then fine tune the settings to suite my needs.

To deploy these containers on the cloud we use Kubernetes [36], which is a container orchestration and management system. With Kubernetes, I can assign volumes for databases and increase the storage size without causing any disruption. It handles scaling by creating multiple instances of a container when it need more resources, and provides service discovery and load balancing for containers [38]. I also make use of Google Kubernetes Engine [37] to run my containers on Google Cloud.

5.7 The Event and Processing Technique Life Cycle

In this section I explain the different workflows that my prototype handles.

5.7.1 Adding a new event

As illustrated in Fig. 5.5, when a user or analyst adds a new event, either via a web app or programmatically, the communication is done via the *Core API* by sending a POST request to the `/events` route and providing it with all the event’s details such as its name and a list of keywords. The information is stored in the database and the *Core API* performs two tasks. First, it creates the event’s Kafka topic, and second, it sends a JSON representation of that event to the `new_event_created` channel.

I have four services subscribed to that channel: *Twitter Collector*, *Keyword Classifier*, *Tweet Archiver*, and *Active Event Listener*. The *Twitter Collector* receives a message from that channel and sends a GET request to the Core API's `/events/terms` route to retrieve a list of keyword of all active events. This list is then submitted to Twitter's Streaming API, and all the incoming tweets are published to the `raw_tweets` queue in Kafka.

Once the *Keyword Classifier* receives the message on the `new_event_created` channel, it communicates with the Core API by sending a GET request to `/events/active` which returns a list of active events. The Keyword Classifer starts consuming data from the `raw_tweets` queue and classifies each tweet for each event by comparing the events keywords against the contents of the tweets. Once classified, the tweets are pushed to each associated event queue.

The *Tweet Archiver* service also receives a message from the channel; it creates a background job to consume the new event's Kafka queue and store its tweets in Cassandra.

Finally, the *Active Event Listener* receives the new event notification and calls the Core API to find any existing analysis techniques by sending a GET request to the `processing_techniques/active` endpoint. If any techniques exists, the system creates a job by sending a POST request to the `/jobs` endpoint in the Core API, and then it iterates through the processing techniques to initiate the processing of the new event by calling a processing technique's URL and sending it a post request. Once an analysis technique receives that request, it spawns a background process that starts first by updating the job's status to "running" via the Core API. After that the technique begins consuming the event and processes its tweets. The results are stored in a database available in the storage layer. As mentioned above, my prototype stored results for all of its processing techniques in Redis.

5.7.2 Deactivating an event

When a user or analyst deactivates an event (see Fig. 5.6), either via a web app or programmatically, the communication is done via the *Core API* by sending a PUT request to the `/events/:id` route and providing it with the event's ID and changing the value of the "active" attribute to false. This information is stored in the database and the *Core API* then performs two tasks. First, it published a JSON message to the event's Kafka topic to indicates that the event is closed. Second, it sends a JSON representation of the updated event to the `existing_event_deactivated` channel.

I have two services subscribed to that channel: *Twitter Collector* and *Keyword Classifier*. When the *Twitter Collector* receives a message from that channel, it sends a GET request to the Core API's `/events/terms` route to retrieve an updated list of keyword for all active events which will not include the keywords for this newly deactivated event. The new list is then submitted to Twitter's Streaming API and all incoming tweets are once again published to the `raw_tweets` queue in Kafka.

When the *Keyword Classifier* receives an event deactivation message, it communicates with the Core API by sending a GET request to the `/events/active` endpoint to receive an updated list of active events. It then resumes consuming data from the `raw_tweets` queue and begins classifying tweets with the updated set of active keywords. Once classified, tweets are sent to the relevant event queues.

As for the *Tweet Archiver* service, it will continue to process tweets for the deactivated event until it processes the `{"status":"closed"}` message. It will then store any remaining tweets in Cassandra and exit the background job for that event.

If there is an analysis technique processing that event's data, it will receive the final message `{"status":"closed"}` on Kafka's queue indicating that the event is no longer active. It will process the remaining tweets, save the results to Redis or another database, and then change its job's status to "finished" using the Core API. Its spawned process is then terminated.

5.7.3 Adding a new Processing Technique during an Active Event Collection

When a user or analyst adds a new Processing Technique (as shown in Fig. 5.7), either via a web app or programmatically, the communication is done via the *Core API* by sending a POST request to the `/processing_techniques` route and providing it with all the details of the technique, such as its name and a functioning RESTful URL. That information is stored in the database and the *Core API* sends a JSON representation of that analysis technique to the `new_processing_technique_created` channel.

I have two services subscribed to that channel: *Active Event Listener* and *Legacy Listener*. The Legacy Listener will not do anything with respect to creating new jobs for active events. Instead, the *Active Event Listener* will call the Core API to find all active events. If any exist, the system creates a job by sending a POST request to the `/jobs` endpoint of the Core API, and then it will iterate through the events list to initiate processing of each event by calling a processing technique's URL and sending a post request. Once an analysis technique receives that POST request, it spawns a background process that starts first by updating the job's status to "running" via the Core API.

5.7.4 Adding a new Processing Technique to existing Historical Events

When a new processing technique is added and there are historical events to process (as shown in Fig. 5.8), the *Legacy Listener* must respond. When it is notified, the service calls the Core API to retrieve all historical events. That is done by sending a GET request to the `events/active` endpoint and setting the field "active" to be false in the request body. If any historical events exists, the system creates a job by sending a POST request to `/jobs` in the Core API, and then it iterates through the events list to initiate processing of each event by calling a processing technique's URL and sending a post request. Once an analysis technique receives the POST request, it spawns a background process that starts first by updating the job's status to "running" via the Core API. After that the technique begins consuming the event and processing its tweets.

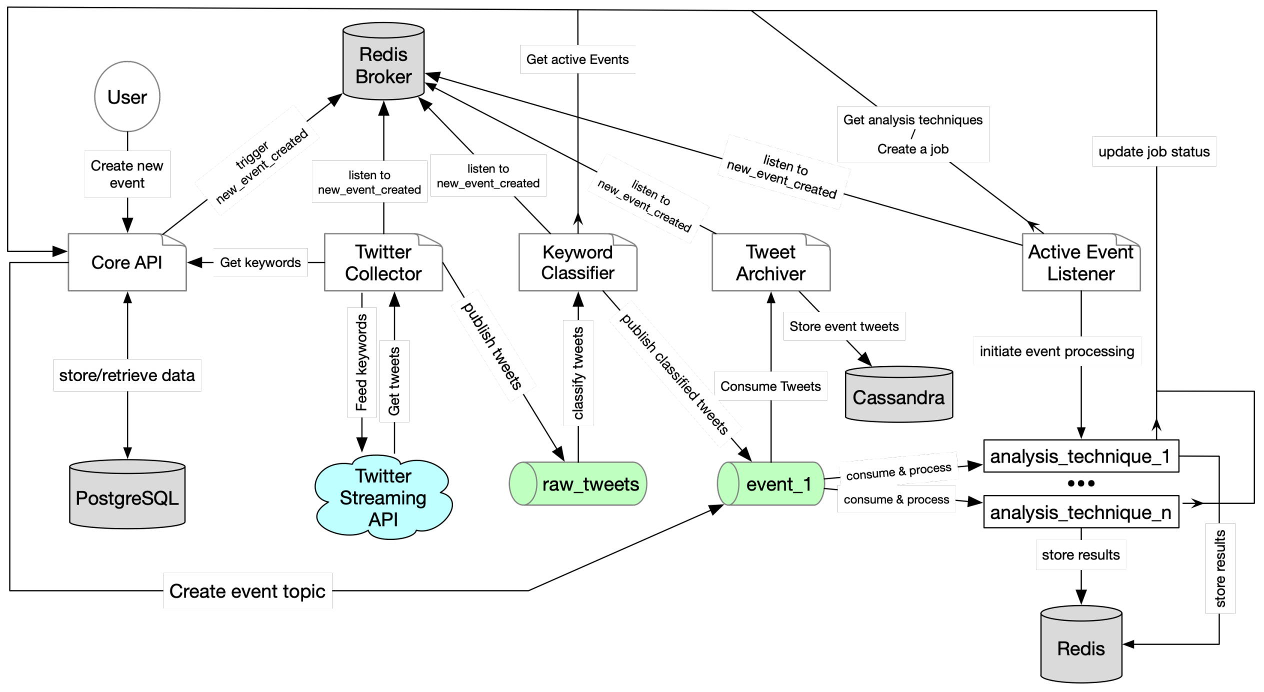


Figure 5.5: The Workflow to Add a New Event

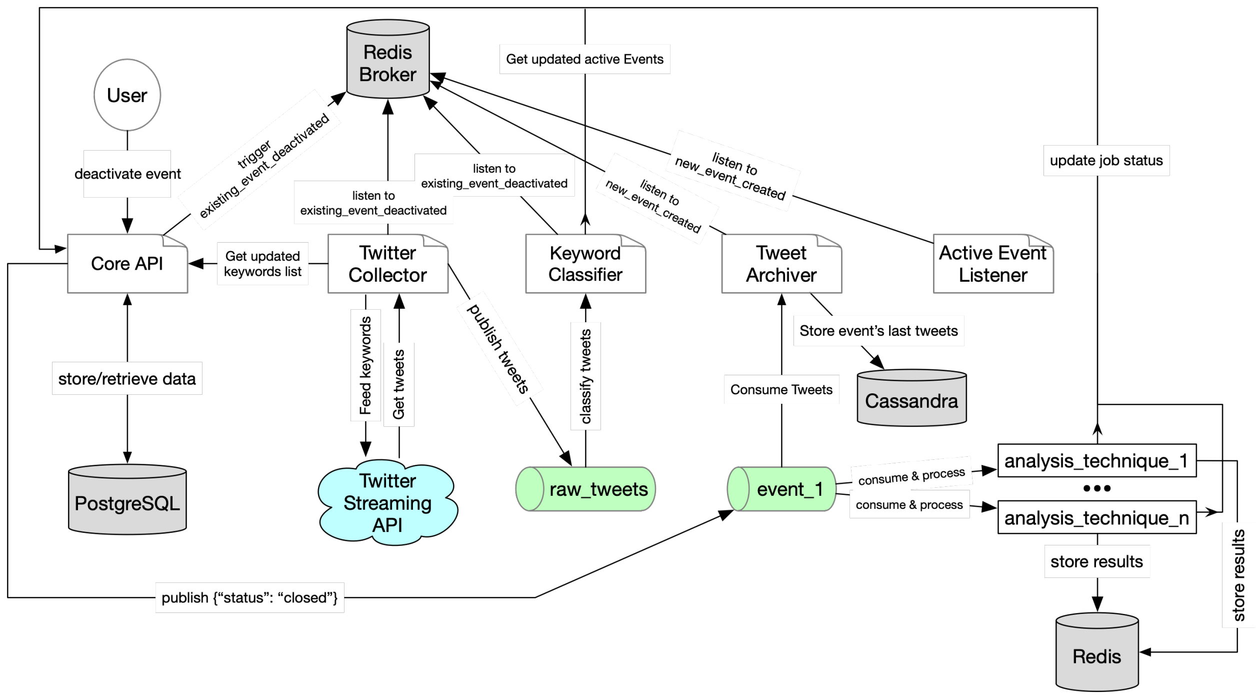


Figure 5.6: The Workflow to Deactivate an Event

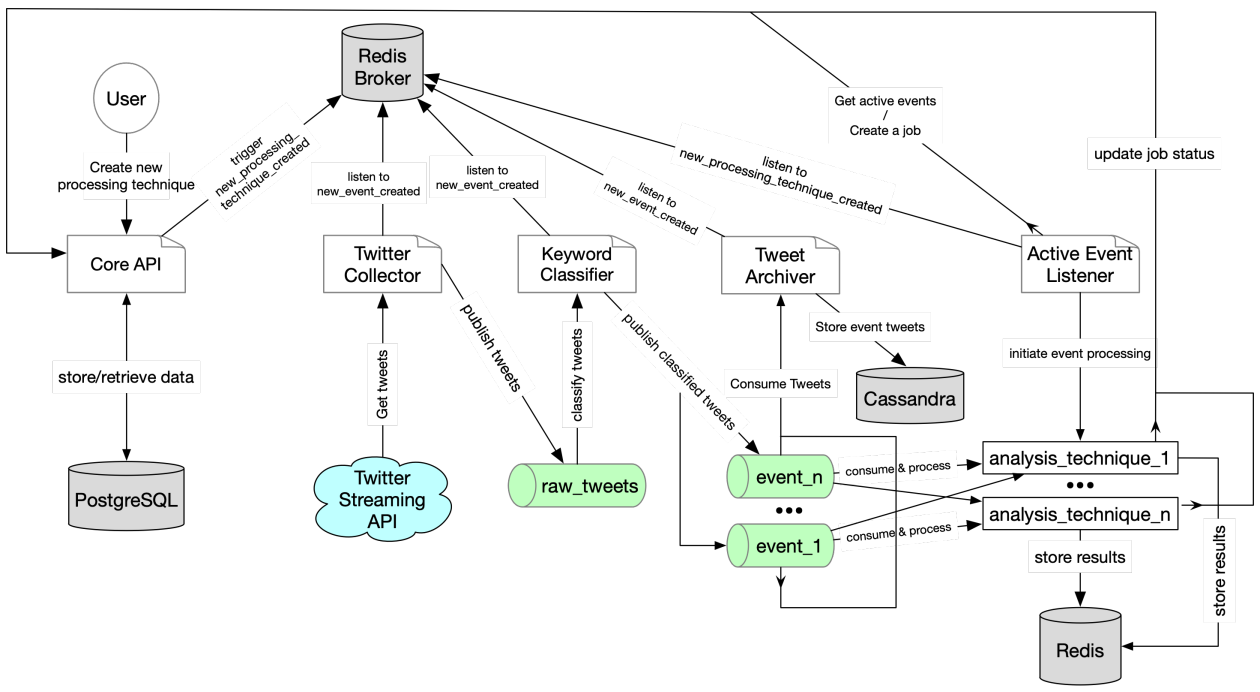


Figure 5.7: The Workflow to Add a Processing Technique for Active Events

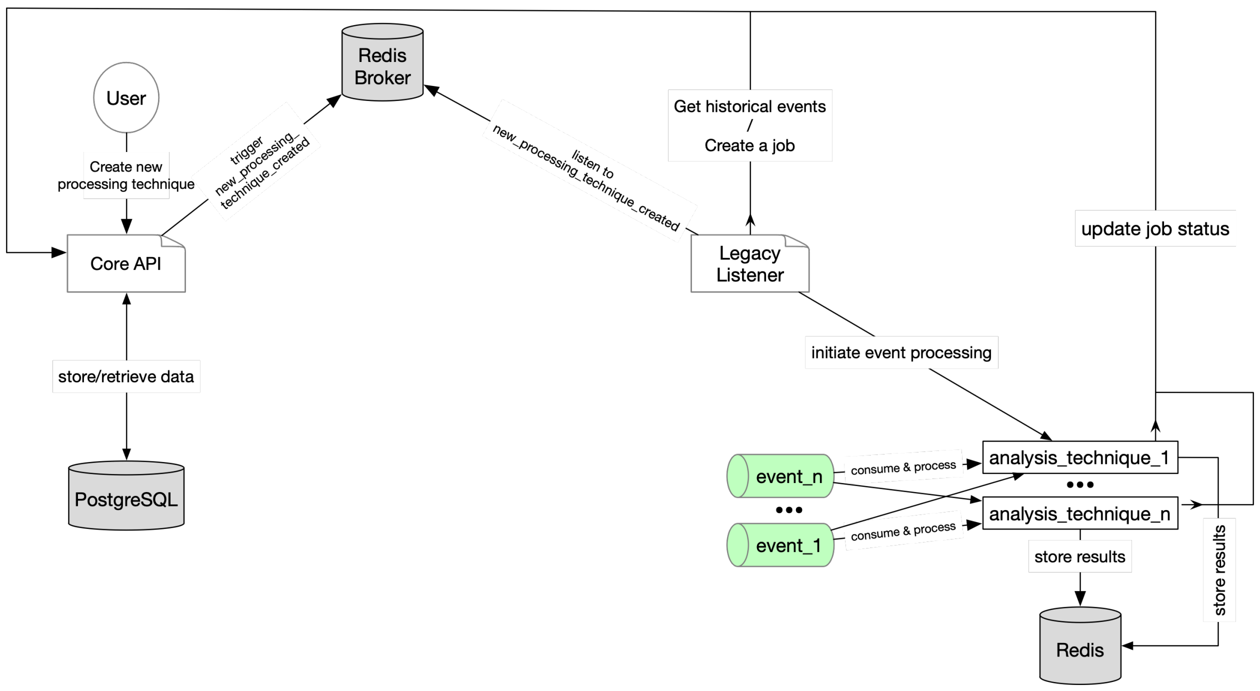


Figure 5.8: The Workflow to Add a Processing Technique for Inactive Events

Chapter 6

Evaluation

This chapter presents the evaluation I performed on my prototype. This evaluation was conducted 1) to test that all analysis workflows described above are implemented correctly and 2) to gain a sense of the performance of the overall system and that my prototype's infrastructure is not unnecessarily slowing down the analysis process.

6.1 The baseline

Before starting to test overall performance of the various services in my prototype, I started with generating a performance baseline of the most basic workflow: collecting a single event and analyzing it with one analysis technique until data collection ends. *For each of my experiments in this chapter, unless otherwise noted, I start with my prototype having no analysis techniques, no events, and no data. That is my prototype starts with a clean slate each time.* Furthermore, the measurement unit used to report results is *tweets per second* for all services.

6.1.1 Active event

To generate my performance baseline for an active event, I started with a clean slate and I added an analysis technique called *Top Hashtags* using the Core API. I then added a new event to the system which starts data collection automatically. The new active event was called "*2019 Republicans*"; it had the following search terms, thought to be associated with very high volume on Twitter: *gop, republicans, republican*, and *trump.* Since *Top Hashtags* was already added, it started processing the new event almost immediately. (There was a slight delay while the Active Listener set up the job to associate it with the new event.) I ran this data collection for 3942 seconds (∼1 hour and 6 minutes). The total amount of tweets for this event was 146,546. I now present the performance of each component of my prototype for this scenario.

The natural flow of the prototype when an event is added is that the *Twitter Collector* starts a collection. In table 6.1, I provide the performance results of Twitter's Streaming API when it collected data using this event's keywords. The number of tweets arriving from Twitter's Streaming API is 38 tweets per second on average.

APP	**RESULTS**		
	Min.	***Max.***	***Avg.***
twitter collector	25	56	38

Table 6.1: Results (tweets/second) for the Twitter Collector service.

After data collection is tweet classification. In the *Keyword Classifier*, each tweet is compared with an event's search term list to check if it matches an event's keywords. Table 6.2 shows the performance when classifying an event; the classification process averages 38 tweets per second. This makes sense: the keyword classifier is essentially classifying tweets as fast as they arrive from the Twitter Collector.

APP	**RESULTS**		
	Min.	***Max.***	***Avg.***
kw classifier	5	49	38

Table 6.2: Results (tweets/second) for the Keyword Classifier service.

The next stage after classification is the execution of an analysis technique to perform analysis on the classified tweets for an event. As mentioned earlier, I used *Top Hashtags* to process this event's tweets. Since this processing technique was added to the system and there was an active event running a collection, the *Active Event Listener* successfully submitted a job to configure the Top Hashtags analysis for this event. As shown in table 6.3, analyzing tweets averaged 36 tweets per second. Also, the amount of tweets processed matched the event's total amount of tweets of 146,546.

APP	**RESULTS**		
	Min.	***Max.***	***Avg.***
top hashtags	2	51	36

Table 6.3: Results (tweets/second) for the Top Hashtags service.

To detect the delay overhead between these services, I compared the logs for each of the above services. The following log messages were generated when the event was deactivated.

The log message for *Twitter Collector*

```
I, [2019-09-19T19:30:40.726907 #1]  INFO -- : existing_event_deactivated: 2019
    Republicans
```

The log message for *Keyword Classifier*

```
I, [2019-09-19T19:30:40.632342 #1]  INFO -- : EVENT -- existing_event_deactivated:
     2019 Republicans
```

The log message for *Top Hashtags*

```
I, [2019-09-19T19:30:43.431285 #14]  INFO -- : 2019_republicans: Finished
    processing 146546 tweets [Top Hashtags]
```

After comparing the timestamps of the *Twitter Collector* and the *Keyword Classifier* services, I concluded that there are no delays. This is because both of these services are subscribed to the `existing_event_deactivated` channel of Redis, which allows them to receive the same message essentially at the same time.

However, I saw a small delay between the *Twitter Collector* and the *Top Hashtags* services when an event is deactivated. The *Twitter Collector* stopped collection for that event at **19:30:40** while the *Top Hashtags* ended processing at **19:30:43**. The difference is only **3 seconds**. This makes sense since the processing technique was still working its way through unprocessed tweets in the queue of its associated event.

6.1.2 Historical event

After the "*2019 Republicans*" event was deactivated, it becomes a historical event with a total of 146,546 tweets. I now wanted to test the behavior and performance of my infrastructure in applying a new processing technique automatically to an existing historical event. I removed the *Top Hashtags* entry via a call to the *Core API* and then added it once more. Adding the processing technique was then detected by the *Legacy Listener* and it created a job to have this new technique process the existing data set. The total time spent processing this historical data set was 73.58 seconds. The performance is, of course, greatly improved (see Table 6.4) since processing a historical event is not limited by the rate in which tweets arrive from the Streaming API.

APP	**RESULTS**		
	Min.	***Max.***	***Avg.***
top hashtags	1200	2607	2004

Table 6.4: Results (tweets/second) for Top Hashtags for the 2019 Republicans as a historical event.

To verify this performance and confirm it as a baseline for this configuration of one historical event with one processing technique, I ran this experiment three times to demonstrate that the performance of the processing technique is stable at around 2000 tweets per second, taking approximately 72 seconds each time. (See Table 6.5.)

EVENT	**RESULTS**			**PROCESSING TIME**
2019 Republicans	***Min.***	***Max.***	***Avg.***	*In Seconds*
1st run	1123	2843	2096	70.34
2nd run	1045	2591	2031	72.31
3rd run	1223	2563	1952	75.14

Table 6.5: Results (tweets/second) for Top Hashtags for the 2019 Republicans event.

I also tested this analysis technique's performance with a very large data set of a historical event exported from *EPIC Collect*. This second event is the "2013 Japan Earthquake" event; it has around 3.4 million (3,482,253) tweets. It took 3126.25 seconds (∼52 minutes) to process this data set. The results shown in Table 6.6 show that processing this event by the *Top Hashtags* processing technique averages around 1100 tweets per second. My system can be scaled to make this processing take less time; I would simply need to configure Kubernetes to invoke additional instances of the analysis technique to process the event's queue in parallel. Since the core use case of my prototype is to process active events, I leave my configuration at just a single instance of each processing technique per event since the processing of active events is limited by the rate at which tweets arrive from the Streaming API. In that primary use case, multiple instances of the processing technique would not lead to significant speed ups.

APP	**RESULTS**		
	Min.	***Max.***	***Avg.***
top hashtags	500	3860	1152

Table 6.6: Results (tweets/second) for Top Hashtags for the 2013 Japan Earthquake event.

After looking at these results and comparing them with table 6.3 where the system was processing an active event, I wanted to confirm that the primary reason for the difference in performance between processing active events and historical events is the Twitter Streaming API. That is, I wanted to make sure that my keyword classifier was not impacting performance.

6.1.3 The bottleneck

To evaluate the performance of the keyword classifier, I decided to temporarily sever the connection between the Twitter classifier and the `raw_tweets` queue and instead pre-load an event into `raw_tweets` and test the performance of just the keyword classifier. I did this by exporting a historical event called the *2012 Casa Grande Explosion* from EPIC Collect. I then loaded the tweets from that event into `raw_tweets` to simulate an active event without any throttling from Twitter's Streaming API. I then used the Core API to create a new processing technique and to create a new event; this sequence then triggered the Active Event Listener to start a new "active" collection. I did this three times (starting with a blank slate each time: no events, no processing techniques, tweets pre-loaded into `raw_tweets`) to evaluate the performance of the keyword classifier (see Table 6.7.)

EVENT	**RESULTS**		
2012 Casa Grande Explosion	***Min.***	***Max.***	***Avg.***
1st run	33	977	488
2nd run	72	971	495
3rd run	6	1014	497

Table 6.7: Results (tweets/second) for the Keyword Classifier for the 2012 Casa Grande Explosion.

As expected, the performance of the keyword classifier in this scenario is much higher than a true active collection, averaging roughly 500 tweets per second across all three runs. Likewise, the processing technique analyzes the tweets as fast as the keyword classifier can generate them (see Table 6.8), also averaging roughly 500 tweets per second. These results show that the components of my prototype add very little overhead to the analysis process. The primary determinants of overall performance will be the Twitter Streaming API for active collections and the processing times of individual analysis techniques. Even if one analysis technique proves to be slow, that is not a showstopper for the overall system. Multiple instances of that processing technique could be launched to help reduce the overall time for processing a particular event and the performance of one processing technique has no impact on the performance of the other processing techniques or on the overall system.

EVENT	**RESULTS**		
2012 Casa Grande Explosion	***Min.***	***Max.***	***Avg.***
1st run	29	1869	508
2nd run	37	2194	504
3rd run	19	2270	522

Table 6.8: Results for Top Hashtags for the 2012 Casa Grande Explosion as an active event.

Furthermore, these results lend strength to my choice of technologies used in my prototype. Kafka queues are designed for being processed in parallel for speed. As mentioned above, if I wanted to boost the performance of my prototype for the processing of historical events, I can simply configure Kubernetes to invoke multiple instances of a processing technique per event queue. For historical events, I would see significant boosts in speed with respect to total processing time (i.e. total processing time would go down) since the only limiting factor would be the performance of the processing technique itself. For the processing of active events, these additional nodes will not provide much benefit since the primary factor in determining performance in that scenario is the rate at which tweets enter the system from the Twitter Streaming API.

6.2 More events and processing techniques

In this section, I document the performance of my prototype when there are multiple events and analysis techniques active at the same time.

6.2.1 One event, two analysis techniques

I began first by testing if one event can be processed by more than one analysis technique. I started a new event called *2019 Republicans* (a distinct collection from the one mentioned in previous examples), and I ran the collection for 10 minutes to collect 23,696 tweets. Before the event was created, I added the *Top Hashtags* processing technique to the system via the Core API.

While the collection was running and already being processed by *Top Hashtags*, I added the *Top Retweets* analysis technique via the Core API approximately four minutes after the collection started. Introducing a new processing technique in the middle of a collection will cause it to first process all previously collected tweets in the event's queue before it is finally current; it will then only process new tweets as they arrive from the Twitter Streaming API. As expected, when the second analysis technique was activated, it processed thousands of tweets per second for the first six seconds and then dropped to a rate that was throttled by the tweets arriving from the *Keyword Classifier* which, of course, come from the Twitter Streaming API via the *Twitter Collector*. As such, its average tweets per second processing time is higher than *Top Hashtags* which was only processing the active event. The results for both processing techniques are shown in table 6.9.

Tables 6.11 and 6.10 show the results for the *Keyword Classifier* and the *Twitter Collector*

APP	RESULTS		
	Min.	***Max.***	***Avg.***
top hashtags	1	90	36
top retweets	3	2278	76

Table 6.9: Processing results (tweets/second) for the 2019 Republicans event.

services respectively; both average around 38 tweets per second, similar to the averages shown in tables 6.2 and 6.1 in the baseline section and also the reason that the average results reported in Table 6.9 for Top Hashtags is roughly the same value.

APP	RESULTS		
	Min.	***Max.***	***Avg.***
twitter collector	23	56	39

Table 6.10: Results (tweets/second) for the Twitter Collector for the 2019 Republicans event

APP	RESULTS		
	Min.	***Max.***	***Avg.***
keyword classifier	23	50	37

Table 6.11: Results (tweets/second) for the Keyword Classifier for the 2019 Republicans event.

6.2.2 Multiple active events, one historical event, and multiple analysis techniques

After the evaluation in the previous section, my prototype now has *2019 Republicans* as a historical event and two active processing techniques, *Top Hashtags* and *Top Retweets*. I decided to deactivate the *Top Retweets* processing technique to test that my system no longer makes use of it when there are new events to process (and that turned out to be true). I then added a new processing technique, *Top Users*, launched three new events (discussed below), and then added one final processing technique, *Top Languages* approximately four minutes later. These actions triggered execution of all of the core workflows of my prototype and set the stage for an extended data collection and analysis session. With respect to workflows, adding the *Top Users* processing technique causes it to start processing the existing historical event (*2019 Republicans*) immediately due to the work of the Legacy Listener. Launching three new events causes the Active Listener to set-up the analysis of each of those events by the three active processing techniques. Finally, adding the *Top Languages* technique four minutes after the start of this experiment, caused the Active Listener to configure this analysis technique to process the three active events and it caused the Legacy Listener to configure it to process the historical event. All of these workflows were activated as expected and ran to completion. Additional details of this scenario are presented below.

Firstly, for this experiment, I configured my prototype to have Kubernetes launch two instances of each of the three processing techniques. As mentioned above, this boosts performance of the processing of the historical event. Secondly, the three new events that I created were:

(1) *2019 Democrats* with terms: [dems, democrat, democrats, democratic party, Elizabeth Warren, Bernie Sanders, joe biden, kamala harris, beto]; this event collected 100,642 tweets.

(2) *2019 Impeachment* with terms: [impeach, #ImpeachTrumpNow, #ImpeachmentTaskForce, #ImpeachmentIsComing, #ImpeachThePres, impeach trump, Impeachment, trump, Pelosi]; this event collected 286,674 tweets.

(3) *2019 Hong Kong Protests* with terms: [hong kong, hong kong protest, Hong Kong protesters, china protest]; this event collected 11,549 tweets.

All three of these events ran for eighty-four minutes. The terms for the *2019 Impeachment* event had a higher frequency on Twitter during the time of my collection, which led to a larger data set being collected for that event than the other two.

Table 6.12 shows the results for all events that were processed by *Top Hashtags*. The average varies depending on the number of tweets collected for each event; The *2019 Hong Kong Protests* event had the lowest number of tweets and therefore had had the lowest average (since all three averages are being computed over the same eighty-four minute data collection period).

As for *Top Users*, when it was introduced before I added the new events, it automatically processed the *2019 Republicans* historical event. Table 6.13 shows its results for all events.

Event	RESULTS		
	Min.	***Max.***	***Avg.***
2019 Democrats	1	46	19
2019 Impeachment	1	69	37
2019 Hong Kong Protests	1	6	2

Table 6.12: Processing results (tweets/second) for Top Hashtags.

Event	RESULTS		
	Min.	***Max.***	***Avg.***
2019 Democrats	1	48	19
2019 Impeachment	1	69	37
2019 Hong Kong Protests	1	6	2
2019 Republicans	3	3001	1911

Table 6.13: Processing results (tweets/second) for Top Users.

Since *Top Languages* was added after the collection had started, it processed a chunk of tweets that had already been collected very quickly and then slowed down once it had caught up and was waiting for new tweets to arrive. This technique also processed the *2019 Republicans* event automatically. Table 6.14 demonstrates its results for all of the events.

Event	RESULTS		
	Min.	***Max.***	***Avg.***
2019 Democrats	1	1009	21
2019 Impeachment	1	1344	39
2019 Hong Kong Protests	1	28	2
2019 Republicans	999	2492	1900

Table 6.14: Processing results (tweets/second) for Top Languages.

For the three new events, the Twitter Collector's performance was approximately the same as for a single event coming in at 48 tweets per second on average. (See Table 6.15.) This rate is, of course, out of my control. It depends on the rate of tweets being generated by Twitter users that match our keywords and any limits being placed on our stream by Twitter.

As for the *Keyword Classifier*, it averaged 46 tweets per second for the entire experiment

APP	**RESULTS**		
	Min.	***Max.***	***Avg.***
twitter collector	1	71	48

Table 6.15: Results (tweets/second) for the Twitter Collector.

classifying tweets for all three active events. (See Table 6.16.) Once again, the performance of the keyword classifier is tied to the performance of the Twitter Collector which, in turn, is constrained by the Streaming API.

APP	**RESULTS**		
	Min.	***Max.***	***Avg.***
kw classifier	1	84	46

Table 6.16: Results (tweets/second) for the Keyword Classifier for all events.

6.2.3 Findings

As a result of my evaluation, I was able to determine that all of the core workflows of my prototype were implemented correctly and functioning as expected. Furthermore, I was able to determine that my prototype does not add any significant overhead to the automation of the analysis of large data sets. The components of my prototype that manage the automation process perform their tasks efficiently leaving just two components that drive the overall performance of the system: the rate of Twitter's Streaming API and the performance of individual analysis techniques. The analysis techniques that I implemented were straightforward and execute quickly but that will not always be true. An analyst may develop an analysis technique that takes a long time to process each tweet or may slow down as it processes more and more tweets. In those cases, the analyst will have to work with my system's configuration parameters to figure out the best way to deploy that technique to mitigate those performance problems as needed. For instance, they might configure Kubernetes to spin up multiple instances of the technique to help it process tweets in parallel (horizontal scaling) or they might deploy the service on a high performance machine with lots of memory (vertical scaling). Either way, my prototype can still incorporate the technique and ensure that it is applied to all tweets from all events, both active and historical.

Chapter 7

Related Work

In this section, I present information on related work in the area of big data software engineering and big data analytics.

In [26], a software architecture for analytics is introduced by the authors to support queries on top of streaming and historical data sets using a Domain Specific Language (DSL). The goal is to present a summary of all collected information in near real time. Their work addresses the three main requirements of "data-driven decision making" for various members and stakeholders in software development projects. The first requirement focuses on *immediacy*, which is providing near real time insight on processed data independent of whether it comes from a historical data set or was just collected seconds before. The second requirement concentrates on designing a DSL as an abstraction layer on top of an existing query engine, such that processing and querying data would be simplified rather than writing ad hoc scripts for extracting and exploring the data, which can be time consuming. The last requirement is about aggregating and summarizing the processed streams of extracted data—both old and new—to provide a high level of overview and insight. Furthermore, in their work, these summaries are tailored to different stakeholders on a software development team. This work is similar to my own with respect to processing both streaming data and already collected data. However, I am not building a DSL to query or transform my data, and I am trying to provide pipelines for different types of data processing techniques.

The work in [56] introduces a cloud-based application that applies queries to Twitter data while it is being streamed, and allows running ad hoc queries against historical and newly streamed data combined for near-real-time data analysis. These queries are generic and answer common questions about Twitter data such as trending hashtags, most retweeted tweets, and so forth. Most importantly, they designed a data model for storing tweets in Cassandra that indexes the collected data to allow for fast queries. The work in [28] is similar but leverages the Lambda Architecture and event-based programming techniques to help analysts in answering questions on social media data collected during crisis events.

In [4], the design of a system to process Twitter data in real-time was presented; in this paper, the focus was on tweets generated by Saudi Arabian users. Sentiment analysis was performed in order to understand how Saudi Twitter users expressed opinions on various public opinion based topics; these opinions can be positive, neutral, or negative and must consider the use of different Saudi dialects. The system made use of Apache Flume to funnel tweets from Twitters Streaming API to Apache Spark. In Spark, it used Spark Streaming and Spark Engine to process and clean the data, and then used a MapReduce based sentiment analyzer that the team developed; this analyzer includes a dictionary of dialects to classify the tweets to have positive, neutral, or negative opinions. Although the architecture is designed to perform only sentiment analysis on tweets, the authors made interesting use of Flume to aggregate and stream data and to use it as a pipeline or channel to transfer data from the streaming API to Spark Streaming.

Another related project is a big data framework called BINARY [19]. It is a SaaS-based framework that performs ad hoc querying to explore and visualize data by integrating various data sources into different kinds of data storage while processing the integrated data with different processing technologies. According to the authors, they use different kinds of database technologies in parallel such that raw and processed data can be stored and indexed using the right kind of database to allow fast ad hoc querying of data. They claim that different data sources can be added due to the use of the parallel databases concept. They also use multiple processing frameworks such as Hadoop, Hive, Impala, and others to perform multiple types of analytics metrics and issue ad hoc querying process. Although the main purpose of this prototype is performing ad hoc queries, the concept of using data processing techniques or frameworks is very close to my work.

Real-time big data analytics systems enable the user to gain valuable insight into streaming data; in [10], researchers examine another field that generates data of this type: eLearning platforms, also known as learning management systems (LMS). In [10], the team designed an architecture to process data being generated from an LMS called Moodle. Generated Moodle data is consumed, processed by running "statistical analysis on the fly," and then stored for later evaluation. With this system, the researchers wanted to gain insight on the frequency of student use and the daily or hourly access of the courses within the LMS. In their software prototype, the researchers use Kafka as a message broker, Flume to ingest data stream from Moodle to Kafka, Spark and Spark Streaming for data processing, and HBase and HDFS for storing the analytical results. Later the result is viewed and visualized on a web interface with D3—a Javascript library for visualizing data as graphs, histograms, pie charts, and other visual data representation styles.

All of these systems make use of similar technologies to address issues and challenges in the domain of big data software engineering. My system is unique in its focus of establishing a generic, extensible platform for performing analysis on large data sets, both historical data sets that had been previously only partially analyzed as well as new data sets under active collection.

Chapter 8

Future Work

As it is human nature to be curious about gaining additional insights on data, there will always be a need to add new features and enhancements to my system. As one example, I currently have implemented only a handful of analysis techniques for my prototype; as such I would like to implement the rest of the analysis features mentioned in Jambi's work [29] in a way that will work with my prototype and with each of them implemented as independent services.

In the future, I would also like to add full-text search capabilities over the events stored in my prototype by integrating either Apache Solr or ElasticSearch. I believe this feature will be straightforward to implement by creating an analysis technique that simply adds tweets to one of these systems automatically. I would then need to build a user interface that would allow requests to be made on the selected system allowing analysts to issue queries and see their results.

I would also like to abstract the Queuing Layer's implementation even further so that it can use other kinds of queuing technologies besides Kafka. This might be implemented by using the Repository design pattern [20] in which the code only deals with the abstraction of publishing and consuming queues without the knowledge of which queuing technology is being used. I think this will make the architecture more extensible, independent, and future proof to new or better queuing technologies in the future.

Furthermore, my work can be expanded by adding a user interface layer to the architecture and implementing a comprehensive and intuitive set of interfaces to the analysis work being performed by my system. Such an interface would make it easy to understand what active and historical events exist and to understand the state of the analysis work being performed on those events. An analyst would be able to see, for example, that their currently active event is being processed by ten analysis techniques and would be able to see how much of the collected tweets for that event had been analyzed; the same would be true of any historical events that they imported into the system or when they added a new analysis technique and it began processing all of the existing historical data sets. A final enhancement would be to then add a bunch of useful information visualization techniques to the user interface, such as pie chart and other graphs to provide views of statistical data in real time, such as tweet volume for a given event or the processing speed of an individual analysis technique.

Chapter 9

Conclusions

My work has led to the design, implementation, and deployment of a software architecture that automates the analysis of actively collected and historical events in support of crisis informatics research.

In my work, I learned from the limitations of previous systems built to support storing and analyzing Twitter data for analysts in the field of crisis informatics. I took advantage of their lessons learned with respect to data models and useful software technologies and took advantage of the results of extensive prior studies designed to make clear the data analysis needs of crisis informatics researchers.

Leveraging this prior work, allowed me to focus on my core issues of automating analysis of large crisis data sets. I designed a software architecture that was heavily influenced by the Kappa Architecture and made extensive use of queuing technologies and microservices. These decisions provided me with an extensible and highly responsive software infrastructure to meet the needs of my users. My system can detect the arrival of new events and the arrival or departure of analysis techniques and do the right thing in response ensuring that (eventually) all existing data sets (both active and historical) have been processed by all active analysis techniques. I designed a framework for creating processing techniques which is generic enough to allow for the integration of a wide range of analysis into my system. These techniques are implemented as services that live separate from my infrastructure and can be scaled and deployed independently and can make use of any set of technologies that they want. With the addition of a user interface layer, my processing technique framework can easily be extended to allow analysis techniques to report on the results of their analysis in a way that would allow for generic visualizations.

The contribution of my work to software engineering is the design of my software architecture and the framework for creating processing techniques that together provide for the creation of scalable and extensible data collection and analysis environments for big data software engineering. The contribution of my work for crisis informatics is the creation of a big data software system that significantly reduces the need for manual analysis of large crisis data sets. The existing set of analysis techniques address a significant number of common metrics that crisis informatics researchers require for their research and ask of each new crisis data set. Now, with my system, these questions are computed automatically with the results available shortly after data collection ends. In the future, with the addition of a user interface layer, the system will represent a nearly complete environment for supporting a wide range of tasks associated with crisis informatics research.

Bibliography

[1] Kenneth M. Anderson. Embrace the challenges: Software engineering in a big data world. In Proceedings of the 2015 IEEE/ACM 1st International Workshop on Big Data Software Engineering, BIGDSE '15, pages 19–25, Washington, DC, USA, 2015. IEEE Computer Society.

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

[3] Kenneth M. Anderson and Aaron Schram. Design and Implementation of a Data Analytics Infrastructure in Support of Crisis Informatics Research (NIER Track). In Proceeding of the 33rd international conference on Software engineering - ICSE '11, ICSE '11, pages 844–847, Waikiki, Honolulu, HI, USA, 2011. ACM Press.

[4] Adel Assiri, Ahmed Emam, and Hmood Al-dossari. Real-time sentiment analysis of Saudi dialect tweets using SPARK. In 2016 IEEE International Conference on Big Data (Big Data), pages 3947–3950. IEEE, dec 2016.

[5] Ahmet A. Aydin. Incremental data collection & analytics the design of next-generation crisis informatics software. PhD thesis, University of Colorado at Boulder, 2016.

[6] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Migrating to Cloud-Native architectures using microservices: An experience report. In Communications in Computer and Information Science, volume 567, pages 201–215. Springer, Cham, sep 2016.

[7] Mario Barrenechea, KennethM. Anderson, AhmetArif Aydin, Mazin Hakeem, and Sahar Jambi. Getting the Query Right: User Interface Design of Analysis Platforms for Crisis Research. In Philipp Cimiano, Flavius Frasincar, Geert-Jan Houben, and Daniel Schwabe, editors, Engineering the Web in the Big Data Era, 15th International Conference on Web Engineering, ICWE 2015, volume 9114 of Lecture Notes in Computer Science, pages 547–564, Rotterdam, The Netherlands, 2015. Springer International Publishing.

[8] Carl Boettiger. An introduction to docker for reproducible research. SIGOPS Oper. Syst. Rev., 49(1):71–79, January 2015.

[9] Vinayak Borkar, Michael J. Carey, and Chen Li. Inside "Big Data management". In Proceedings of the 15th International Conference on Extending Database Technology - EDBT '12, pages 3–14, New York, New York, USA, mar 2012. ACM Press.

[10] Abdelmajid Chaffai, Larbi Hassouni, and Houda Anoun. E-Learning Real Time Analysis Using Large Scale Infrastructure. In Proceedings of the 2nd international Conference on Big Data, Cloud and Applications - BDCA'17, pages 1–6, New York, New York, USA, 2017. ACM Press.

[11] Jürgen Cito, Philipp Leitner, Thomas Fritz, and Harald C. Gall. The making of cloud applications: an empirical study on software development for the cloud. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015, pages 393–403, New York, New York, USA, aug 2015. ACM Press.

[12] CORBA® BASICS. `http://www.omg.org/gettingstarted/corbafaq.htm`. Accessed: 2017-10-11.

[13] DataStax: always-on data platform. `https://www.datastax.com/`. Accessed: 2017-10-18.

[14] Distributed Component Object Model. `https://technet.microsoft.com/en-us/library/cc958799.aspx?f=255{&}MSPPError=-2147217396`. Accessed: 2017-10-11.

[15] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM, 51(1):107–113, jan 2008.

[16] David J DeWitt, Robert H Gerber, Goetz Graefe, Michael L Heytens, Krishna B Kumar, and M Muralikrishna. GAMMA - A High Performance Dataflow Database Machine. In Proceedings of the 12th International Conference on Very Large Data Bases, VLDB '86, pages 228–237, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[17] Philippe Dobbelaere and Kyumars Sheykh Esmaili. Kafka versus RabbitMQ. In Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems - DEBS '17, pages 227–238, New York, New York, USA, 2017. ACM Press.

[18] What is a Container. `https://www.docker.com/resources/what-container`. Accessed: 2019-10-09.

[19] Azadeh Eftekhari, Farhana Zulkernine, and Patrick Martin. BINARY: A framework for big data integration for ad-hoc querying. In 2016 IEEE International Conference on Big Data (Big Data), pages 2746–2753. IEEE, dec 2016.

[20] Eric Evans. Domain-driven design: tackling complexity in the heart of software. Addison-Wesley, Boston, 1 edition, 2004;2003;.

[21] Roy T. Fielding and Richard N. Taylor. Principled design of the modern Web architecture. In Proceedings of the 22nd international conference on Software engineering - ICSE '00, pages 407–416, New York, New York, USA, jun 2000. ACM Press.

[22] Martin Fowler and James Lewis. Microservices. `http://martinfowler.com/articles/microservices.html`, 2014. Accessed: 2015-11-13.

[23] Shinya Fushimi, Masaru Kitsuregawa, and Hidehiko Tanaka. An Overview of The System Software of A Parallel Relational Database Machine GRACE. In Proceedings of the 12th International Conference on Very Large Data Bases, volume 86, pages 209–219, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[24] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. ACM SIGOPS Operating Systems Review, 37(5):29, dec 2003.

[25] Seshadri Gokul. J2EE: What It Is and What It's Not. `http://www.informit.com/articles/article.aspx?p=26660`, 2002. Accessed: 2017-10-11.

[26] Georgios Gousios, Dominik Safaric, and Joost Visser. Streaming Software Analytics. Proceedings of the 2nd International Workshop on BIG Data Software Engineering - BIGDSE '16, pages 8–11, 2016.

[27] Apache Hadoop! `http://hadoop.apache.org/`. Accessed: 2017-09-16.

[28] Sahar Jambi and Kenneth M Anderson. Engineering Scalable Distributed Services for Real-Time Big Data Analytics. In 2017 IEEE Third International Conference on Big Data Computing Service and Applications (BigDataService), pages 131–140. IEEE, apr 2017.

[29] Sahar H. Jambi. Engineering Scalable Distributed Services for Real-Time Big Data Analytics. PhD thesis, University of Colorado at Boulder, 2016.

[30] M. S. V Janakiram. How to deploy microservices to derive maximum benefit. Computer Weekly, March 2015. pp. 14–17.

[31] Xiaolong Jin, Benjamin W. Wah, Xueqi Cheng, and Yuanzhuo Wang. Significance and Challenges of Big Data Research. Big Data Research, 2(2):59–64, feb 2015.

[32] Apache Kafka - A distributed streaming platform. `https://kafka.apache.org/intro.html`. Accessed: 2019-10-09.

[33] Kappa Architecture - Where Every Thing Is A Stream. `http://milinda.pathirage.org/kappa-architecture.com/`. Accessed: 2017-10-17.

[34] Jay Kreps. Questioning the Lambda Architecture. `https://www.oreilly.com/ideas/questioning-the-lambda-architecture`, 2014. Accessed: 2017-10-17.

[35] Jay Kreps. It's Okay To Store Data In Apache Kafka. `https://www.confluent.io/blog/okay-store-data-apache-kafka/`, 2017. Accessed :2019-10-09.

[36] Kubernetes Production-Grade Container Orchestration. `https://kubernetes.io/`. Accessed: 2019-10-09.

[37] KUBERNETES ENGINE Reliable, efficient, and secured way to run Kubernetes clusters. `https://cloud.google.com/kubernetes-engine/`. Accessed: 2019-10-09.

[38] What is Kubernetes? `https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/`. Accessed: 2019-10-09.

[39] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review, 44(2):35–40, 2010.

[40] Sylvain Lebrense. A thrift to CQL3 upgrade guide. `https://www.datastax.com/dev/blog/thrift-to-cql3`, 2012. Accessed: 2017-10-18.

[41] Ling Liu. Computing infrastructure for big data processing. Frontiers of Computer Science, 7(2):165–170, apr 2013.

[42] Bernard Marr. Why only one of the 5 Vs of big data really matters — The Big Data Hub. `http://www.ibmbigdatahub.com/blog/why-only-one-5-vs-big-data-really-matters`, 2015. Accessed: 2015-11-13.

[43] Nathan Marz and James Warren. Big Data: Principles and Best Practices of Scalable Realtime Data Systems. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2015.

[44] Casey Mctaggart. Analysis and Implementation of Software Tools to Support Research in Crisis Informatics. Master's thesis, University of Colorado at Boulder, Jan 2012.

[45] Definition of Big Data by Merriam-Webster. `https://www.merriam-webster.com/dictionary/big%20data`. Accessed: 2017-11-01.

[46] Sam Newman. Chapter 1: Microservices. In Building Microservices, chapter 1, pages 1–11. O'Reilly Media, 1 edition, 2015.

[47] Leysia Palen and Kenneth M Anderson. Crisis informatics–New data for extraordinary times. Science, 353(6296):224–225, jul 2016.

[48] Leysia Palen, Kenneth M. Anderson, Gloria Mark, James Martin, Douglas Sicker, Martha Palmer, and Dirk Grunwald. A vision for technology-mediated support for public participation & assistance in mass emergencies & disasters. Proceedings of the 2010 ACMBCS Visions of Computer Science Conference, pages 1–12, 2010.

[49] Leysia Palen, Sarah Vieweg, Sophia B. Liu, and Amanda Lee Hughes. Crisis in a Networked World. Social Science Computer Review, 27(4):467–480, nov 2009.

[50] Mike P. Papazoglou and Willem-Jan van den Heuvel. Service oriented architectures: approaches, technologies and research issues. The VLDB Journal, 16(3):389–415, jul 2007.

[51] Postgresql: The world's most advanced open source relational database. `https://www.postgresql.org/`. Accessed: 2019-10-16.

[52] Redis. `https://redis.io/`. Accessed: 2019-10-09.

[53] Samza. `http://samza.apache.org/`. Accessed: 2017-10-17.

[54] Aaron Schram and Kenneth M. Anderson. MySQL to NoSQL: data modeling challenges in supporting scalability. In Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity - SPLASH '12, SPLASH '12, pages 191–202, Tucson, Arizona, USA, 2012. ACM Press.

[55] Jack Shemer and Phil Neches. The Genesis of a Database Computer. Computer, 17(11):42–56, nov 1984.

[56] R.M. Simmonds, P. Watson, J. Halliday, and P. Missier. A Platform for Analysing Stream and Historic Data with Efficient and Scalable Design Patterns. In 2014 IEEE World Congress on Services, pages 174–181. IEEE, jun 2014.

[57] Apache Solr. `http://lucene.apache.org/solr/`. Accessed: 2017-09-16.

[58] Apache Spark - Lightning-Fast Cluster Computing. `https://spark.apache.org/`. Accessed: 2017-09-16.

[59] Kate Starbird. Examining the Alternative Media Ecosystem Through the Production of Alternative Narratives of Mass Shooting Events on Twitter. In ICWSM, pages 230–239, 2017.

[60] Michael Stonebraker and Lawrence A. Rowe. The postgres papers. Technical report, University of California at Berkeley, Berkeley, CA, USA, 1987.

[61] Apache Storm. `https://storm.apache.org/`. Accessed: 2017-09-16.

[62] Robbie Strickland. Making the Change from Thrift to CQL (Cassandra Query Language). `https://academy.datastax.com/planet-cassandra/making-the-change-from-thrift-to-cql`. Accessed: 2017-10-18.

[63] Johannes Thones. Microservices. IEEE Software, 32(1):116–116, jan 2015.

[64] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. Building a replicated logging system with apache kafka. Proc. VLDB Endow., 8(12):1654–1655, August 2015.