TODO

A DISSERTATION PROPOSAL SUBMITTED TO MY COMMITTEE
IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

PHD

IN

INFORMATION AND COMPUTER SCIENCES

By

Anthony J. Christe

Dissertation Committee:

Philip Johnson and Milton Garces, Chairperson
todo

September 15, 2015
Version 0.0.3

[todo]

ABSTRACT

[todo]

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

1.1 The Big Data Problem

Data is being produced at an unprecedented rate worldwide. Not only is it servers, computers, and mobile phones producing data, but also an impressive amount of heterogeneous sensors and systems that keep our infrastructure running, the Internet of Things (IoT). In 2014 Gartner estimated over 3.7 billion internet connected devices were in use and that by 2020 over 25 billion internet connected devices will be use.

In January 2014, President Obama asked his administration to perform a 90 day study on big-data and its impact on education, national security, government, private industry, and privacy [**?**].

The result of this study yielded two reports. The first report, *Big-Data: Seizing Opportunities, Preserving Values* [**?**], was prepared by the President's Executive Office and focuses on big-data policy and how big-data changes the relationships between government, citizens, education, and the private sector.

The second study by The President's Council of Advisors on Science and Technology (PCAST), *Big-Data and Privacy: A Technological Perspective* [**?**], was prepared to complement the first study and examines the technological and privacy implications of big-data.

Many types of physical phenomena can be modeled as waveforms and digitized by sensor networks. For example, networks of seismometers monitor vibrations in the Earth's crust. Weather stations are constantly taking atmospheric measurements. Vast networks of ocean buoys are measuring wave height. Power quality sensors are measuring power quality inside peoples' households. Infrasound sensors are monitoring volcanoes, large atmospheric events, and other large explosions. Our households, vehicles, and lives are full of sensors. Many of these sensors require near-real-time responses. All of these sensors are distributed in both time and location. The wealth of information is incredible and we're only now starting to engineer the tools needed to digest it.

Physical phenomena modeled as waveforms presents us with three challenges. The tools required to handle a high volume of data, the ability to perform real-time quality analysis and filtering, and the ability to automatically classify transient events found in the waveform both locally and distributed over many sensors.

1.2 Framework

I present a framework for the real-time classification and analysis of transients in geographically and temporally distributed sensor data. We utilized machine learning, graph, and big data algorithms

in order to meet the complex criteria of multi-variant heterogeneous data classification. Further, we utilize cloud architecture and big-data compute frameworks as a means to digest large amounts of heterogeneous data without needing to setup our own hardware.

By utilizing actor systems we are able to maintain and scale to thousands of streaming sensor connections. Actors also allow us to pipeline our initial acquisition process for quality control and binary decoding based on message type and API version of our data. Certain actor models allow us to pass directly into a computer framework of our choice, i.e. Akka actors can pipe data directly into Apache Spark.

By utilizing a compute framework similar to Hadoop or Apache Spark, we are able to injest data and perform batch waveform analysis over time scales of our choosing. Certain frameworks such as Apache Spark Streaming providing sliding window processing on top of real-time streaming data.

My framework provides and integrates many tools for performing waveform analysis on top of a cluster of computers. We utilize Apache Spark's core libraries to perform machine learning and graph processing. We utilize the Python libraries numpy, SciPy, obspy, and Matplotlib to build on top of open source libraries that already provide digital signal processing routines. Many of these open source libraries are also compatible with large compute frameworks such as Apache Spark.

We provide an interface for exploring and performing ad-hoc data analysis on real-time and batch waveform data. We allow users to enter parameters for their specific data set, and then provide the tools needed to access raw data as well as data products for their specific use case.

Our framework is built on top of a cloud architecture utilizing Amazon Web Services (AWS). AWS provides scalability by making it easy to duplicate and increase resources on the fly. We show that by seperating our framework components into seperate instances of AWS servers, we are able to scale coponents on demand and individually.

My framework takes steps to maintain users' privacy and ensure anonymity.

Finally, we show that these techniques provide a measurable increase in performance over traditional methods of sensor data collection and analysis by feeding the framework with sensor data from temporally and geographically distributed power quality sensors and infrasound sensors.

This thesis builds upon the guidelines and ideas laid out by the White House big-data papers. Specifically, we place a heavy emphasis on privacy.

1.2.1 What is Big Data?

Today's technologically driven society has given way to the "Internet of Things", a world wide network of sensors all taking measurements, analyzing and creating massive amounts of data. These massive data sets are collectively referred to as "Big Data".

The White House's report on big-data describes three key points that differentiates big-data from "small data": volume, variety, and velocity.

The volume of data being produced has far surpassed anything produced in history. The internet has allowed for the ubiquitous interconnection of devices which allows almost anything to communicate over a network. It's estimated that in 2013 4 zettabytes of data were generated world wide [**?**]. This is up from 295 exabytes in 2007 [**?**], a $1,256\%$ increase data in just 6 years. In 2010, Google produced the same amount of content (most of it user generated) in two days as was produced from the dawn of time up until 2003 [**?**].

The variety of data being produced is also increasing. Not only are humans sharing more data about themselves (photos, videos, etc), but metadata is also being generated from that data. Furthermore, the Internet of Things provides for sensors that can measure almost anything imaginable. The types of data being transmitted are very diverse.

The variety of data can be attributed to how data is produced. The PCAST paper describes data as either born digital or born analog. Born digital data is a product created specifically for digital consumption, i.e. e-mail, figures, or mouse clicks. Born analog data is often collected from sensors including microphones, barometers and thermostats among others. Born analog data is digitized and then acted upon.

1.2.2 Big Data Context

Our research involves the aggregation of distributed time series data and the detection of deviations from the steady state of sensor signals. Our sensors are either mobile or stationary. In certain configurations, such as the collection of infrasound data using an array of microphones, more information can be gathered when sensors are near each other and physically arranged in a certain situation. Using graph processing algorithms, we show that it is possible to create adhoc mobile arrays.

Our data is time and location sensitive. Our calculations depend on the physical location of sensors (namely latitude and longitude) as well as accurate timestamps to synchronize data streams from multiple sensors.

Infasound Data

Infrasound is sound which is less than 20 Hz in frequency. Infrasound can be detected with both microphones and barometers. We've developed a next generation sensor network for the collection

of infrasonic data. These sensors are distributed, mobile, and make use of constant network connections either over a cellular radio network or over WiFi. Each sensor collects and computes several metrics which when combined with other sensor metrics can be used to compute the location, energy yield, and time of infrasonic events.

Infrasound is generated from many sources, but the particular events we are interested in are large atmospheric events. These can be caused by large explosions, volcanic eruptions and volcanic gas venting, sonic booms, large storms, debris entering the atmosphere, tsunamis, nuclear weapons, etc. Many of these events are time sensitive and early warnings and quick classification could potentially save lives.'

Traditional infrasound arrays utilize a set of stationary microphones configured to "listen" to certain frequency bands. An array is able to determine direction and location of infrasonic events by measuring the differences of when events were picks up on each microphone.

We show that it's possible to use graph algorithms to create ad-hoc arrays from roaming dense sensor networks. By collecting all the data from each device, we can even perform this analysis ex post facto at different frequency bands.

We perform quality control on our data by utilizing decision trees. [This needs some expanding still].

We utilize passive and active training to create templates of waveform transients in-order to classify transients in near real-time.

[This section needs a little updating] For microphone data, each infrasound sensor records one minutes worth of audio data, decimates the audio data by 1000, and then speeds it up a factor of 60. This audio data is stored in a canonical wave file along with metadata associated with the sensor. Other than the normal wave header, the metadata we store is the api version, the idForVendor field from iOS, latitude, longitude, altitude, speed, horizontal accuracy, vertical accuracy, time of solution, and time of last sample.

Each microphone reading we receive contains approximately 9,700 bytes of data once a minute. We can estimate the data we would receive at scale using the following chart.

Sensors	GB/day
1	0.013008714
10	0.13008714
100	1.3008714
1000	13.008714
10000	130.08714
100000	1300.8713
1000000	13008.714

Power Quality Data

We measure consumer grade power quality (PQ) data using an array of devices distributed across Hawaii. In the United States, the standard voltage of electricity is is 120V and the standard frequency is 60 Hz. Other countries have other power standards. Deviations from these standards are considered PQ issues. Common issues in power quality are voltage swells, sags, and frequency fluctuations. By analyzing the raw waveform of electricity, it's possible to detect and classify these PQ issues.

PQ may have large impacts on the lifetimes of electronic devices. Further, the high density of photovoltaics (PV) interacting with a power grid can cause unknown effects. By observing PQ events across the grid in a distributed and time sensitive fashion, we show it is be possible to track points of failure and to provide possible PQ predictions and modeling.

Having quick access to these metrics would be an invaluable resource for power companies, city governments, manufactures, and households.

The high sample rate necessary for detecting short lived transients in power systems requires a large amount of data to be sent across the network. Intelligent triggering can help alleviate some of the congestion, but a new framework is needed in-order to handle the large amount of data.

We utilize machine learning to classify PQ events and decision trees to assign quality metrics to the data received from our distributed sensor network. We use a real-time compute framework such as Apache Spark Streaming to classify local and distributed events in real-time.

Graph algorithms allow us to localize events and define ad-hoc PQ neighboorhoods.

Any Data

Although this thesis focuses on two sources of data that I've collected, it's meant to be an abstract framework that can work with any source of heterogeneous temporally and geographically distributed data. We provide machine learning for transient analysis and quality control. We use graph algorithms for geographic feature extraction. All backed by a large cluster compute framework which can handle the demanding needs of tomorrow's IoT and Big Data explosion.

1.3 Big Data and Privacy

1.3.1 Privacy Issues with Born Analog and Born Digital Data

[todo]

1.3.2 Content and Use

[todo]

1.4 Big Data and International Laws

[todo]

CHAPTER 2
RELATED WORK

2.1 Dealing with big-data

The PCAST report identifies several technologies that have proven use when analyzing big-data sets. These technologies can be broken up into data processing frameworks and distributed cloud providers.

2.1.1 Distributed Cloud Services

Distributed cloud providers offer scalable solutions for provisioning large amounts of processing power, network bandwidth, and storage solutions (both volatile and non-volatile). Cloud providers will often replicate data across multiple geographically distributed physical machines. Access of data stored on these providers will then use the closest server to their access point. By replicating the data across multiple servers, cloud services are also able to provide a large amount of redundancy and uptime for critical applications. The PCAST report mentions several public cloud providers including Amazon Web Services, Google Cloud Platform, Microsoft Azure, and Rackspace.

2.1.2 Data Processing Frameworks

Cloud services only provide half of the equation for performing big-data analytics. There are a multitude of frameworks that provide APIs and algorithms for performing large scale data analytics. These frameworks are able to perform calculations and store data across hundreds if not thousands of machines as if they were a single machine. Some of the most popular solutions include Google's MapReduce (where Hadoop [**?**] is the standard open source implementation), not Structured Query Language (noSQL) databases, Apache Accumulo [**?**], Berkeley Data Analytics Stack [**?**], Google's Dremel [**?**], MPI [**?**] (and OpenMPI [**?**]), OpenMP [**?**], Pregel [**?**], Cloudscale [**?**], and Apache Spark [**?**].

MapReduce / Hadoop

Google's MapReduce is a programming model for creating and processing large data sets. The MapReduce framework can access data from both file systems and structured data sources such as local and distributed databases. This type of framework works best with embarrassingly parallel data. That is, data where local calculations can be made without knowing the state of the entire system.

First, a map function is applied to local data on each machine within the distributed cluster. This map function transforms the local data into a partial result. Finally, the reduce algorithm takes the partial results from each local node to produce a final result.

An open source version of MapReduce was released to the Apache Software foundation as Hadoop. Hadoop applications can be ran on many of the distributed cloud services mentioned above.

Berkeley Data Analytics Stack

The Berkeley Data Analytics Stack is a collection of open source components built by Berkeley's AMPLab for the purpose of "making sense of Big Data". This framework contains many pieces for working with big-data including:

1. Mesos - Cluster resource manager
2. Hadoop Distributed File System (HDFS)
3. Hadoop Map Reduce
4. Tachyon - Distributed Memory Centric Storage System
5. Spark - Memory optimized execution engine
6. Spark Streaming - Spark framework for streaming applications
7. GraphX - Graph network computations
8. MLbase - Machine learning engine
9. Shark - SQL API
10. BlinkDB - MySQL with bounded errors and response times
11. Hive - Distributed DB engine
12. Storm - Distributed and Real-Time Computation Engine
13. MPI - Message Passing Interface for low level distributed computing

Cloudscale

Cloudscale is a service that attempts to make cloud systems which are scalable by design. This framework allows programmers to program without considering scalability and Cloudscale handles the details of resource management and auto-scaling as the service demands.

Cloudscale provides remoting in which applications look like regular applications to the programmer, but to the Cloudscale network, applications are able to remotely communicate with each other behind the scenes for efficient data sharing. Cloudscale also provides virtual machine management as well as application monitoring. Finally, Cloudscale provides easy deployment of applications to the cloud.

MPI and OpenMP

MPI and OpenMP are both low level messaging APIs that allow processes and threads to communicate with each other with ease. In a sense, these APIs are similar to MapReduce in that local processes compute a local result and then use message passing to collect local results and find a global result. MPI tends to provide lower level access where data must be strictly managed. OpenMP provides programming language constructs which attempt to automatically parallelize computations over multiple machines or multiple devices. These APIs often form the backbone of other big-data processing frameworks.

NoSQL, Apache Accumulo

[todo]

2.2 Sensor Time Synchronization

2.2.1 Introduction

Infrasound, oftentimes referred to as low-frequency sound, is characterized by sound that has a frequency lower than 20Hz. The detection of infrasonic data has traditionally been done using expensive stationary sensor arrays. We've developed an inexpensive solution to create a large heterogeneous distributed sensor network using microphones available on certain mobile devices.

Our sensor network communicates with our acquisition and analysis servers over the built-in WiFi of the mobile devices as well as over cellular based data networks. These sensors are distributed globally and display large deviations in upload and download latency. These deviations in latency introduce errors into our timing accuracy.

In-order to perform array processing, our infrasound data files must be within one sample of actual time. In this paper, we examine what it takes to get our timing within one sample of accuracy. We explore the affects of statistically filtering data based on latency symmetry, hardware turnaround time, and overall latency. We show that this is possible and describe the algorithms we use to obtain this level of accuracy.

[PQ importance here]

2.2.2 Background

Timing and synchronization is an important area of large scale distributed sensor networks. Distributed sensor networks often have unique requirements which are not found in other time sensitive applications. Restrictions on energy use, computational ability, storage, density of sensors, and the distributed nature makes normal timing solutions unsuitable for large scale distributed sensor networks [**?**].

Further, hardware clocks are imperfect and may drift over time. With an increasing density of nodes in a network brings forth an increasing error of clock drifts on each individual node. Clock synchronization is needed to provide a common time base for all nodes in a sensor network where timing is critical. This is especially important when dealing with sound based data where a common time base is required for tracking the source of a sound and for correlating the same sound event picked up on multiple sensor nodes [**?**].

F. Sivrikaya and B. Yener [**?**] define clock synchronization is the following way. Real time can be represented by a clock $C(t)$, but since clocks contain drift and offset factors they're better represented by $C_i(t) = a_i t + b_i$ where $a_i(t)$ is the drift and b_i is the absolute offset from actual time. Two clocks within a network can be compared with the equation $C_1(t) = a_{12} * C_2(t) + b_{12}$ where a_{12} is the relative drift between the two clocks and b_{12} is the relative offset between the two clocks. When the relative drift between two clocks is 1 and the relative offset is 0, its said that those clocks are perfectly synchronized. Two types of synchronization in a global network exist:

global synchronization where all clocks in the network share a common time base and local clock synchronization where clocks (usually those closest to each other spatially) share a common time base. For our needs, we implemented a solution using local clock synchronization. This concept of synchronization can further be generalized into three subclasses [**?**]. The first and most relaxed class only defines the order in which events are ordered temporally. That is for each event e_n it's only necessary that its known that e_1 comes before e_2 comes before e_3 and so on. The second class of synchronization algorithms requires that nodes don't change their individual clocks, but rather the drift and offset are known for all clocks or a subset of clocks in the network. This allows the true time for any clock in the network to be calculated. The third class of synchronization requires that all clocks be synchronized to a common time base at all times. This is the most difficult class of synchronization to implement in practice.

[GPS]

[NTP]

2.2.3 Sensor Message Exchange

Our algorithm builds on top of the Tri-Message exchange sensor exchange algorithm proposed by Tian et al [**?**]. Tri-Message was designed to provide clock synchronization for high-latency resource-constrained networks such as underwater acoustic networks on long range space communication. The main tenants of the algorithm are only a small number of message exchanges are needed for synchronization, can work in high latency networks, simple methods (computationally) for finding the drift and offset. The Tri-Message algorithm makes two assumptions. First that the drift rate is relatively constant for each node and second that the latencies during a single message exchange remain constant. We found in practice that over our networks, high variations in latency introduced large amounts of error into our calculations. We discuss how we got around this in later sections by applying statistical filtering based on latency.

Each node in our sensor network collects 4096 samples of audio data before transmitting the data to our data acquisition server. Our nodes sample data at 80 Hz which means that each node in our network sends data approximately every 51.2 seconds. While the audio data is being recorded, each node attempts to exchange 10 "Tri-Message" exchanges with either it's local timing synchronization server or with the fallback global timing synchronization server if a local server is unavailable. All message exchanges for a local node for a single audio file are then embedded in the header of the audio data so that post-facto synchronization can occur once the data has been collected by the data acquisition server.

A single message exchange between a node B and a synchronization server called the anchor A contains six parts. Three microsecond timestamps represented as the number of microseconds since the epoch (January 1st, 1970 UTC) and three microsecond machine time timestamps which represent the internal crystal oscillator of a node since the device has been powered on. In certain

circumstances, the device id is also transmitted to the anchor time server. The message exchange takes place as described in figure [fix add figure] with each A coefficient representing an anchor epoch timestamp and each B coefficient representing a node machine time timestamp.

Notable differences from Tian et al. Tri-Message exchange include an initial $B0$ from the node which initializes the synchronization process on the anchor server as well as a $B4$ message from the node which includes the device id of the node. The device id is useful when the timing coefficients are stored on the anchor server rather than the device nodes. The device id allows the anchor server to identify which node the anchor server is performing synchronization with.

Once a message exchange has occurred, it's possible to correct the nodes internal time by finding the drift β and offset α with the following equations

$\beta = (B3 - B1)/(A3 - A1)$

$\alpha = (B1 + B2)/2 - (A1 + A2) * \beta/2$

2.2.4 Constraints on Latency

Due to the fact that latencies have a high variance in practice, we've implemented a statistical solution to estimate the true time $A0$ from a series of message exchanges. Since each audio file contains 4096 samples, it's possible to find a single audio file whose timing coefficients meet our statistical criteria and then correct the rest of the audio files adjacent to it using a known sample rate and the number of samplers per audio file. This allows us to essential "lock-on" to a good file and use the timing information for that file to perform a post-facto timing synchronization on all files adjacent to that "locked-on" file.

The statistical values that we are looking for are all variations on the latency. We look at overall latency, symmetric latency, and turn-around time on nodes and anchors.

The latency can be estimated with two different sets of timing coefficients. We can estimate the latency by taking the round-trip time of A and subtracting the turn-around time of B or by taking the round-trip time of B and subtracting the turn-around time of A. The two latency equations can be defined as:

$d1 = ((A2 - A1) - (B2 - B1))/2$

$d2 = ((B3 - B2) - (A3 - A2))/2$

Turn-around time is defined as the time it takes for a device to receive a timestamp, generated a new timestamp, and send the new timestamp back to the sender. This can be defined as:

$T_A = A3 - A2$

$T_B = B2 - B1$

Symmetry looks at the latency from the anchor server to the node and from the node to the server. The Tri-Message algorithm requires that inbound latency be similar to outbound latency. We can test the symmetry of the latency with the following equations:

$S = (B3 - B1) - (A3 - A1)$

To find the actual time for a given file, we first filter our files using the timing coefficients and the latency equations above. We perform filtering in an order that will remove the most noise first and then move on to filters that will disqualify a smaller number of message exchanges. Our filtering process using the above filters in the following order: node turn-around (T_B), anchor turn-around (T_A), symmetry (S), and finally overall latencies ($d1, d2$). We filter message exchanges using these latencies with the following values.

Once we've filtered out the good message exchanges, we can find the actual time $A0$ using the following equations:

$$A01_i = A1_i + d1_i - (B1_i - B0_i)$$
$$A03_i = A3_i + d3_i - (B3_i - B0_i)$$
$$A0_{mean} = \sum(A01_i + A02_i)/(2 * numMessages)$$

2.2.5 Experimentally Finding Latency Filters Values

2.2.6 Conclusion

2.2.7 Figures

2.3 Berkeley Data Analytics Stack

2.3.1 Apache Spark

Spark [**?**] is a general purpose computation engine for large-scale data analytics. Spark runs on top of Hadoop and makes use of Hadoop's distributed file system. Spark is implemented in Scala and provides public APIs for Python, Java, and Scala. The framework also provides an interactive interpreter through Python and Scala. Spark replaces Hadoop's MapReduce engine with highly efficient in-memory computations. Spark supports applications with similar scalability and fault tolerance requirements as MapReduce. Spark claims speed increases of up to 100 times that of Hadoop. Several Spark components play an important role within the Berkeley Data Analytics Stack, namely the basic computation engine as well as the Spark's streaming framework. Since Spark is built on top of Mesos, a cluster operating system, Spark can share data and work with other distributed frameworks such as Hadoop or MPI.

Apache Spark focuses on two areas of computation that MapReduce and its variants does not provide. Iterative computations such as those found in machine learning algorithms and interactive analytics. Apache Spark differs from MapReduce by providing the ability to store and cache large data sets in main memory allowing future computations to easily make use of past computations. By chaining and pipelining processes together, Apache Spark achieves high performance on large commodity clusters.

The Spark programming model revolves around writing a driver program to implement high-level logic and to launch parallel operations. Data is defined in terms of *resilient distributed datasets* (RDD) and then these datasets are transformed in parallel by passing functions over the data sets. The scheduling of these computations is handled at the framework level by Spark.

RDDs represent an abstract collection of lazy objects which are read-only and partitioned across multiple compute units. RDDs contain fault tolerance mechanisms such that if any RDD partition is lost, it can be recomputed from the Spark compute history. RDDs can be cached, shared, and accessed from any part of the framework during computation. Being an abstraction, RDDs can be created from many data sources including files, Scala and Java collections, or through the transformation of other RDDs.

RDDs can be transformed in parallel using operations such as *map*, *reduce*, *filter*, *collect*, and *foreach*. Map takes a user defined function and transforms each data item using that function. Reduce applies a function over associative data using the previous function's result as an input into the next computation. Collect collects the results from each parallel operation and aggregates those results within the driver program. The foreach parallel action is similar to *map* in that it maps a function over the data, however this function is mainly used to for the generation of side effects (as opposed to purely functional view) from the function.

Apache Spark supports two types of shared variables which can be accessed from any thread.

Broadcast variables which are read only and the preferred approach will disseminating information across workers and accumulator variables which can be used for adding associative values across workers.

2.3.2 Apache Spark Streaming

Spark Streaming [**?**] is a project from Berkeley's AmpLab which support real-time parallel processing of streams of data. Spark Streaming aims to improve upon previous batch computation frameworks in three important ways. Fault tolerance and proper replication of data across a cluster, consistency of data across multiple workers in a cluster (do they see the same data?), and unification of batch processing (i.e. the ability to perform operations using real-time data and historical data at the same time). Spark Streaming is built on the principals of Spark in that it's main abstraction of data is an RDD. Spark Streaming solves the above issues by adding the concept of discretized streams (D-Streams) which are a series of RDDs that can be acted on in parallel using similar operations as Apache Spark (map, reduce, filter, etc).

D-Streams are used to compute batch clusters of data over small time intervals. Data received this way is distributed and stored across workers on the cluster. Once enough data has been received for a given time-interval, batch computations are applied to the data. D-Streams are operated on using sliding window algorithms and multiple windows may be used concurrently to analyze different time intervals. Artifacts and intermediate data can be computed this way and stored back to the cluster as RDDs. D-Streams also support incremental aggregation (such as computing running statistics) and time-skewed joins such that previously computed RDDs in the stream can be used with current data. This effectively allows to different time intervals to be inspected.

D-Streams are fault tolerant and use a parallel recovery as an approach to recreating lost partitions in Apache Spark. They track the lineage of a stream, and if any part of the stream is lost, it can be rebuilt using its lineage.

2.3.3 Spark SQL

Spark SQL was originally developed as Shark SQL [**?**] and later merged into the Spark project as a core library. Spark SQL was designed to support both SQL and complex analytics efficiently. Spark SQL performs in-memory computations using the same RDD data structures used in Spark and Spark Streaming. The ability to use RDDs allows Spark SQL to perform complex analytics including machine learning algorithms and graph processing. The use of RDDs also facilitates the same fault tolerance guaranties as other data partitioned across the cluster.

Spark SQL adds three extensions to Spark to provide for more efficient SQL queries.

2.3.4 GraphX

GraphX [**?**]

2.3.5 MLBase

MLBase [**?**]

CHAPTER 3
SYSTEM IMPLEMENTATION

3.1 Software Architecture

We make heavy use of BDAS components to analyze our data. Data from sensors are streamed to our server and aggregated using Spark Streaming. Data is streamed to Spark Streaming where it is transformed into a discrete stream of Spark RDD data structures. Real-time processing acts on these RDDs and performs one of three actions.

3.1.1 Apache Spark Streaming

First, data is partitioned and cached in-memory. We use an aging algorithm to get a different granularity and density of data depending on how long it's been sitting in-memory. For example, data from the last N minutes may be stored with a high density of detail and information. Data from the last M hours will be stored in a more compact format at the cost of losing detail. Data from the past P days could be stored with the lowest information density. These measurements will remain in memory so that the data can be analyzed at various time scales.

Second, real-time processing actively seeks out transients and deviations from the steady state of the signal. Another way of looking at it is that the real-time processing will filter out uninteresting data when the signal remains at steady state. When a deviation is encountered, the system can communicate with other devices for smart triggering, perform additional analysis, and store interesting information to the SQL database.

Third, the real-time processing unit is responsible for storing "interesting" data at an appropriate level of detail on the SQL database. Some of this data may be artifacts ready ready for consumption and some of the data will be detailed raw data for further future analysis using Apache Spark at a future date.

3.1.2 Apache Spark

Once data has been pre-processed by Streaming Spark, we can schedule Apache Spark jobs to run longer term analysis on the stored in the aged temporal in-memory data. Apache Spark also provides interpreters in Java and Python that allow us to dynamically query the data when we are iterating

3.1.3 Adhoc Mobile Arrays

We use AmpLab's GraphX framework for performing parallel graph computations on our mobile and stationary sensors.

3.1.4 Front End

3.2 Data Flow

APPENDIX A
APPENDIX: [TODO]

[todo]