Data Management for Distributed Sensor Networks: A Literature Review

Anthony J. Christe

February 16, 2017

Abstract

Sensor networks can benefit from the generally "unlimited resources" of the cloud, namely processing, storage, and network resources. This literature review surveys the major components of distributed data management, namely, cloud computing, distributed persistence models, and distributed analytics.

Contents

List of Figures

1 Introduction

The exponential increase in volume, variety, velocity, veracity, and value of data has caused us to rethink traditional client-server architectures with respect to data acquisition, storage, analysis, quality of data, and governance of data. With the emergence of Internet of Things (IoT) and increasing numbers of ubiquitous mobile sensors such as mobile phones, distributed sensor networks are growing at an unprecedented pace and producing an unprecedented amount of streaming data. It's predicted by the European Commission that IoT devices will number between 50 to 100 billion devices by 2020[51].

The size of sensor networks is quickly growing. BBC Research provides figures that the market share for sensor networks in 2010 was $56 billion and was predicted to be closer to $91 billion by the end of 2016 [63]. Data generated from the IoT are surpassing the compute and memory resources of existing IT infrastructures. [17]. Not only is the size of data rapidly exploding, but data is also becoming more complex. Data from sensor networks is often semi-structured on unstructured with data quality issues.

Sensor networks can benefit from the generally "unlimited resources" of the cloud, namely processing, storage, and network resources. We believe that by leveraging cloud computing, distributed persistence models, and distributed analytics, it's now possible to provide a platform that is able to meet the demands of the increasing distributed sensor market and the increasing volume, velocity, variety, and value of data that comes along with that.

This review summarizes the current state of the art surrounding distributed sensor networks and the use of cloud computing as a means for big sensor data acquisition and analysis. In particular, we will define Big Data and review it in the context of sensor networks, review cloud computing and service models related to distributed sensing, discuss modern distributed persistence for Big Data, and modern distributed analytics for Big Data all with an emphasis on acquiring and managing Big Sensor Data.

1.1 Applications of Distributed Sensor Networks

Zaslaveky et al. [63] cites several examples of distributed sensor networks in-the-wild including: a real-time greenhouse gas detection network deployed across California, real-time structural monitoring such as the St. Anthony Falls Bridge sensor network in Minneapolis, distributed radiation detection in Fukushima, real-time parking space inventory in San Francisco.

Perera et al. in their paper on sensing as a service[48] provide three examples of areas distributed sensor networks would accelerate at.

First, distributed sensors could be used by cities to optimize waste management which consumes a significant amount of time, money, and labor. Waste management also has many processes including collection, transport, processing, disposal, and monitoring. By collecting and storing sensor data in the cloud from these processes, various interested parties could access sensor data in

order to optimize for the current state of the system. As an example, Perera mentions that city council members could optimize garbage routes and collection rates based on the amount of trash available and recycling centers could forecast what to expect based off of the same sensor data. Basically interested parties at all points of the management process could benefit by analyzing data points from IoT devices in a smart city.

Second, Perera mentions that smart agriculture can take advantage of distributed sensor networks and cites the *Phenonet* project as an example of distributed agricultural sensing which has the ability to monitor plant growth, soil composition, air composition, and pests. A major advantage of this system is that it can supplement traditional research by allowing multiple researchers access to the same data in near real-time.

Third, Perera postulates that environmental management could utilize existing distributed environmental sensors upgraded to communicate with the cloud allowing for data sharing and data fusion among interested parties.

Gerla et al.[25] propose an internet of vehicles as a means to autonomous vehicles. By treating vehicles as platforms of thousands of sensors each and by creating dynamic distributed clouds, they hope to allow fleets of vehicles to make autonomous decisions. This model uses distributed clouds based on proximity and peer-to-peer to technologies rather than sending data to a centralized cloud. The real-time nature and the size and amount of sensors makes this an interesting case study.

One area that shows a lot of promise for distributed sensor networks with centralized management is smart grids. The smart grid is an collection of technologies aiming to advance the electrical grid into the future with respect to intelligent energy distribution and integration of renewable. Electrical grids can benefit by using a large distributed sensor network to collect power consumption, production, and quality information and use that information to control power production and consumption in real-time.

In some cases, the sensor nodes in smart grids lack powerful local computation abilities, but generally have network connections and sensing capabilities. This makes the cloud a perfect sink of information for analyzing complex power trends from a large scale distributed sensor network for smart grids[11].

1.2 Rest of this Review

The rest of this review is structured as follows: Section 2 provides an overview of Big Sensor Data2. Section 3 will focus on cloud computing and how its concepts can be utilized to manage distributed sensor data. Section 4 will examine the current state of the art distributed persistence models with an emphasis on how NoSQL and distributed persistence models can aid in managing distributed sensor data. Section 5 will examine the current state of big data analytics options in the cloud and how these can be utilized for performing analytics on distributed sensor data.

2 Big Data

Big Data is described using many definitions. Cox, in 1997[18], provides us with one of the earliest definitions where Big Data is “too large to be processed by standard algorithms and software on the hardware one has available to them”. He also mentions that sources for big data collections include data from remote sensors and satellite imaging in the fields of atmospheric sciences, geophysics, and healthcare.

Cox separates Big Data into big data into two categories; namely, *big data collections* and *big data objects*.

Big data objects are single, very large data sets such as computational models computed from physical phenomena. Big data objects often do not fit in memory or local disks. Big data objects also have adverse affects on bandwidth and latency. Cox looks to moving computation to the data and more advanced segmentation and paging techniques at the OS level to deal with big data objects.

Big data collections contain many smaller objects or even many big objects. Big data collections present their own set of issues including: distributed data, heterogeneous data formats, no platform independent definition, non-local meta-data, large storage requirements, poor locality, and insufficient network resources.

Cox provides us a useful definition to build on. He also advocates for the development and advancement of operating system constructs for moving data that is too large for memory in and out of memory using stenciling, segmentation, paging, and application controlled segmentation. It’s interesting to note that this was before cloud computing and distributed systems, but we are now facing similar problems at the distributed level rather than a local level.

The Apache Hadoop project, in 2010, defined big data as “datasets which could not be captured, managed, and processed by general computers within an acceptable scope”[17].

Manyika et al[42] in 2011 define big data as “the amount of data just beyond technology’s capability to store, manage, and process efficiently” essentially making the definition of big data a moving target that is constantly evolving as technology becomes updated.

The Whitehouse report on big data[47], in 2014, defines big data as ”data that is so large in volume, so diverse in variety or moving with such velocity,that traditional modes of data capture and analysis are insufficient” and

Hashem et al.[29] build on these previous definition in their 2015 review on Big Data providing the definition by attempting to create a definition that encompasses the spirit of many of the previous definitions. They define big data as “a set of techniques and technologies that require new forms of integration to uncover large hidden values from large datasets that are diverse, complex, and of a massive scale”.

NIST, in 2015, [46] provide multiple definitions relating to big data. NIST defines big data as “extensive datasets–primarily in the characteristics of vol-

ume, variety, velocuty, and/or variability–that require a scalable architecture for efficient storage, manipulation, and analyst. To my knowledge, NIST is the only organization to specify the need of a scalable architecture alongside its definition of big data. NIST next defines the big data paradigm as "the distribution of data systems across horizontally coupled, independent resources to achieve the scalability needed for the efficient processing of extensive datasets".

Perhaps one of the most popular definitions of Big Data is characterizing data by "the four Vs"[29], *volume*, *variety*, *velocity*, and more recently, *value*.

2.1 The Three (Four and Five) "V's"

The first mention of the three V's was in Laney's 2001 article *3-d data management: controlling data volume, velocity, and variety*[39]. Laney describes the challenges of managing e-commerce data by categorizing the data challenges into three dimensions. First, we will review Laney's definition of the 3 V's and then we will examine updated, expanded, and more modern interpretations of the three V's and also look at the more recent "fourth V".

Volume is the amount of data flowing into a system at any one time. Laney argues that the increased availability of the internet to anyone as an e-commerce platform greatly increases the amount of transactional data stored on server backends. Since data is a tangible asset, organizations may be reluctant to discard the data. At the same time, as the amount of data increases, each individual data point becomes less important. Laney mentions that if organizations are not willing to simply buy more online storage, that they can take the following steps to limit volume growth: implement tiered storage systems, limit data collected to only data required for current organization processes, limit analytics to statistically sampled data, eliminate redundancy in data sources, offload "cold spots" to cheaper storage (i.e. tape), and outsorce data management.

In terms of the increase in data volume, we can look at several statistics starting from the year 2011 and working onwards. According to Gantz et al.[24], the world wide accumulation of data in 2011 was around 1.8 zettabytes. Then in 2013, during the D11 conference, Meeker presented that this figure had risen to 5 zettabytes across the globe. In 2014, more than 500 million photos were uploaded every day and more than 200 hours of video per minute[47]. Tweets generate 12 terabytes of data per day[54]. Perera et al.[48] expect with IoT, we could see as many as 1 billion sensors online and generating data by the year 2020. Power meters generate upwards of 350 billion readings annually[54]. According to IBM[4], "90% of the world's data has been created in the past two years".

Velocity is defined by Laney as "increased point-of-interaction (POI) speed and, consequently, the pace data used to support interactions and generated by interactions", or more generally, the pace of data arriving and how long it takes to analyze, store, and act on that data. Laney offers several solutions to data velocity in using operational data stored that prioritizes production data, front-end caching, point-to-point data routing protocols, and architecting

software in such a way that balances data analysis latency with stated real-time requirements. Some examples of high velocity data include GPS tracking data, web site click streams, social media interactions, and data from mobile sensors[47]. Sharma et al.[54] also mention that over 5 million trade transactions must be processed daily for fraud detection.

Laney describes data variety as data that is in “incompatible formats, non-aligned data structures, and inconsistent data semantics”. Laney’s proposed solutions to variety include profiling data to find inconsistencies, a standardized XML data format, interprocess application communication, middlewares on top of “dumb data” to provide meaning and intelligent, metadata management, and more advanced indexing techniques. The main reason for the increase in variety of data is due to the prevalence of internet connected devices and the internet of things (IoT) explosion. We now see a wide variety of data that was *born analog* such as sensors measuring our physical world like temperature, solar radiance, power quality, seismic activity, acoustics, etc. We also see much more variety in *born digital data* from the web, social media, government databases, geospatial data, surveys, healthcare, etc[47].

Value is often added as a fourth ”V” and represents the value that can only be gained by finding insights into big data. Manika et al. in their McKinsey report[41] describe the value of big data after studying the results of long running big data healthcare projects. Value in terms of efficiency and quality of data can be gained from big data using cross correlations and data fusion to gain insights that was not possible before big data. Examples include recommendations from Amazon or Netflix, predict market demand, improve healthcare, and improve security[43]. Chen et al.[17] believe that value is actually the most important V for big data in that big data often has hidden values that can be extracted using big data analytics. Sharma et al.[54] mention that businesses are more and more heavily investing into big data because of the hidden values that could exist.

Finally, veracity has been mentioned alongside the other V’s[54]. As data volume, variety, and velocity increase, there is a fear that the quality of the data may be hard to ascertain or quantify. Thus, veracity is a measure of the trustworthiness of the data.

2.2 Features of Big Data

NIST[46] provides us with a list common features of big data. One common feature of big data is associated metadata. Metadata is data about data that includes information about how/when/where the data was collected and processed. Metadata describing the history of data provides for data provenance. This becomes more important as data is transfered between many processes with multiple transformations. Provenance provides a means to track how data was transferred and how it was transformed. Semantic metadata is an attempt at providing metadata with the ability of describing itself. Examples of semantic metadata include the Semantic Web[10] and NIST’s Big Data Paradigm.

Another common feature of big data is that it can often be unstructured, semi-structured, or non-relational data. Examples of these types of data include unstructured text, audio, video, and other natural phenomenon that create digitized signals from physical samples. We will review in great detail the big data persistence models in section 4 and big data analytical models in section 5 which will examine storage and analysis of unstructured and non-relational data.

Big data sets tend to exhibit their own features. NIST categorizes big data sets into two categories, *data at rest* and *data in motion*.

At rest data is data that has already been collected and is stored in cold storage for non-realtime analysis. The defining feature of data at rest in relation to big data is its volume. It's estimated that by the year 2020, there will be 500 times more data than there was in the year 2011. Big data sets often do not fit on a single server and can be spread out over multiple data centers. Another feature of big data at rest is variety. Data sets can contain data in multiple formats from different domains that in some way need to be integrated to provide value and meaning. These features of data at rest give rise to the need for distributed big data persistence including shared-disk file systems, distributed filesystems, distributed computing, and resource negotiation.

In motion data is processed in real-time or near real-time in order to provide immediate feedback. Examples of data in motion include event processing systems from distributed sensors. In motion data can come in the form of data streams. The main feature of data in motion is its velocity. That is the quantity of data that is required to be acquired, persisted, analyzed, and acted upon is large compared to the time window that these operations need to take place. The amount of data that is needed to be acted on in a time window is too much for a single system and has given rise to parallel distributed computing architectures.

Sensor data is a type of big data that has its own defining features which future complicate acquisition, persistence, and analysis [17]. Sensor data often highly correlates both time and location, producing geospatial timeseries data. One of the most common characteristics of sensor data is the amount of noise in the data. Environmental sensing will always include noise because sensor data is born analog[47]. Often sensor networks provide data in a large variety of unstructured formats with missing, partial, or conflicting meta-data. Sensor data can also contain a large amount of data redundancy from multiple sensors in similar locations. The problem very quickly becomes a needle-in-the-haystack problem, or more aptly stated, finding the signal in the noise.

Chen shows other examples of big data in figure 1.

2.3 Examples of Big Data

Industrial equipment and engines are making use of distributed sensors that generate automated alerts when maintenance is needed[22].

Medicare and Medicaid are using big data predictive analytics to flag fraud

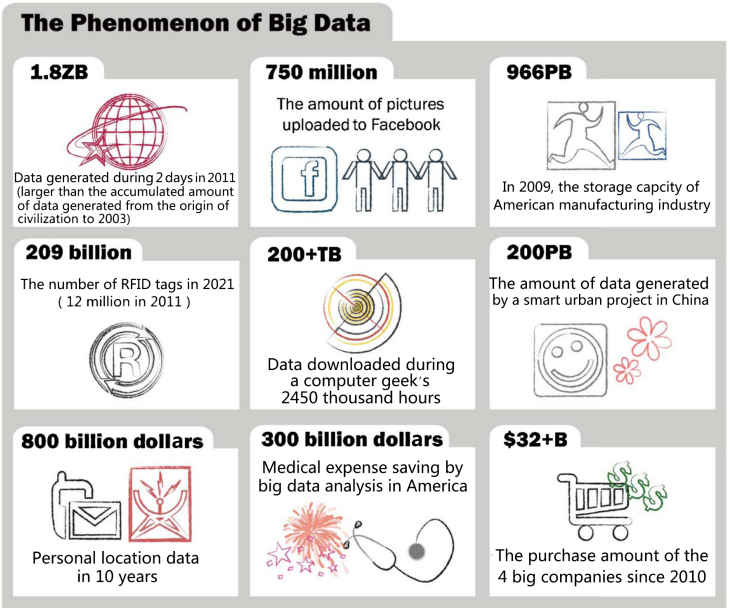

Figure 1: The Phenomenon of Big Data.[17]

before claims are paid to individuals. This has saved over $115 million dollars a year in fraudulent payout[2].

Defense Advanced Research Projects Agency (DARPA) has funded various projects for visualization battlefields in real-time and visualization and creating models for traffic flow through road networks, providing valuable intel on where to locate roadside explosives[47].

Researchers at the Broad Institute were able to detect genetic variants in DNA related to schizophrenia [47]. The interesting thing about this case is that the variants were not discovered until a large number of samples were analyzed. At low numbers of sample, the variant can not be seen. At intermediate numbers of samples there is a small signal. The genetic variants become very clear as soon as a certain threshold of data is obtained.

In a review on smart cities and big data authored by Hashem et al.[28], the authors review many technologies surrounding big data and how big data can play a role in smart city infrastructures of the future. They found the following areas could benefit from using Big Data in a smart city.

Smart grids can improve energy generation efficiency by monitoring environmental data, analyzing the power habits of users, and measuring consumption from smart meters[38]. Smart grids can also make use of big data to perform forecasting of future load generation[6].

Healthcare is another sector that can gain insights on the back of big data. One healthcare project monitored and cross correlated sensors in a neo-natal intensive care unit in order to identify factors that could lead to an infection

or early warning signs of infections. Data that was collected included temperatures and heatrates. Analysis allowed doctors to make diagnosis that they would have missed otherwise without big data analytics[15]. Analytics of big data in healthcare using big data mining techniques can be used for diagnosing patients, predicting illnesses, and predicting epidemics[52].

Smart cities can make use of big data to decrease traffic congestion and better plan freight management by analyzing real time traffic sensors and using predictive analysis to determine traffic routes ahead of time[34]. Of course, Google and other companies already do this by analyzing mobile devices to determine and predict traffic congestion.

Hashem et al.[28] cite several examples of succesful smart city projects in Stockholm, Helsinki, and Copenhagen.

3 Cloud Computing

NIST[45] defines cloud computing as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction".

The major five tenants of cloud computing as defined by NIST are as follows:

On-demand self-service where the user can provision network, storage, and compute capacity automatically without the need for human intervention. In essence, this becomes a virtual shopping mart where to the consumer it appears that virtually unlimited cloud resources are available to chose from and the user (or algorithm) can increase or decrease the utilization of cloud resources at any time.

Broad network access where computation capabilities are performed over a network and results are delivered to clients such as mobile devices.

Resource pooling where resources within a cloud such as storage, network, or compute capacity are shared among multiple tenants. This allows for efficient utilization of hardware when generally virtual services are provided to clients. Clients don't necessarily know where their physical hardware is located or provisioned.

Rapid elasticity is the ability to provision or remove cloud resources (i.e. storage, network, or compute resources) at any time from a system as demand on that system either increases or shrinks. Often times a human may not even be involved in making these decisions and this scaling will take place automatically using a set of predefined usage thresholds.

Measure service where cloud providers provide a means of metering the compute resources that are used by clients. This provides a transparent means of selling cloud computing resources to clients and clients can always know how much capacity they have consumed.

Even though the NIST definition is starting to show its age, its major tenants are still the underlying foundation of cloud software even today. Many additional service and deployment models have been developed since NIST defined cloud computing, but an understanding of the basic underpinnings is required before exploring the rest of this vast filed.

Cloud computing frameworks can provide on-demand availability and scaling of virtual computing resources for storage, processing, and analyzing of very large data sets in real-time or near real-time. This model makes it possible to build applications in the cloud for dealing with Big Data sets such as those produced from large distributed sensor networks.

By using the cloud as a central sink of data for our devices within a sensor network, it's possible to take advantage of central repositories of information, localized dynamic computing resources, and parallel computations. With the advent of cheap and ubiquitous network connections, it's becoming easier to do less processing within sensor networks and to offload the work to a distributed set of servers and processes in the cloud[35].

Cloud computing includes both technical and economical advantages as discussed in [11].

On the economical side, computing resources are pay-per-use. Businesses can dynamically increase or decrease the computing resources they are currently leasing. This makes it possible to utilize massive amounts of computing power for short amounts of time and then scale back resources when demand isn't at its peak. Before cloud computing these same businesses would be required to manage and maintain their own hardware for peak load without the ability to dynamically scale their hardware if the peak load were to increase.

On the technical side, the localization of computing resources provides for a wide variety of benefits including energy efficiency, hardware optimizations, software optimizations, and performance isolation.

3.1 Cloud Computing Service Models

When discussing cloud computing, it's useful to understand the service models that traditional cloud computing provide. The three major service models as defined by NIST[45] are *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS) and *Software as a Service* (SaaS).

At the lowest level is the Infrastructure as a Service (IaaS) model which provides virtual machines that users have the ability to deploy and manage. Users can install operating systems on these virtual machines and interact with deployed virtual machines as if they were local servers. Consumers using IaaS have the ability to manage and provision virtual hardware and network resources, but do not need to worry about the underlying hardware or network infrastructures. Other than providing virtual resources, consuming utilizing IaaS still require a decent amount of systems administration knowledge develop, deploy, and secure applications into the cloud using IaaS.

Sitting in the middle of the traditional cloud service models is the Platform as

a Service (PaaS) model. In this service model consumers don't have the ability to interact or provision individual cloud resources such as virtual machines, storage, networking, or compute capacity. Instead, users have the ability to deploy their application to the cloud via custom cloud provides tools or via a cloud provided application programming interfaces (APIs).

At the highest level is the Software as a Service (SaaS) layer. Generally speaking, applications in a SaaS environment are generally provided by the cloud provider. In a SaaS model, users do not have the ability to control their own cloud resources and users do not have the ability to upload their own applications to the cloud. Users do sometimes have the ability to alter the configuration of the software they are interacting with in this model.

Since the original service models were penned, there have been many other types services models introduced. Several of these focus on IoT service layers as noted by Botta et al's[11]. These include *Sensing as a Service* (S^2aaS), *Sensing and Actuation as a Service* (SAaaS), *Sensor Event as a Service* (SEaaS), *Sensor as a Service* (SenaaS), *Data Base as a Service* (DBaaS), *Data as a Service* (DaaS), *Ethernet as a Service* (EaaS), *Idendity and Policy Management as a Service* (IPMaaS), and *Video Surveillance as a Service* (VSaaS).

Some of the above mentioned service models are of particular interest for a survey examining cloud computing and sensor networks. We will examine these in more detail in sections 3.2.

3.2 Sensing as a Service

Sensing as a Service (SaaS or S^2aaS) describes "the process of making the sensor data and event of interests available to the clients respectively over the cloud infrastructure"[19].

The sensing as a service model includes 4 layers[48]. Figure 2 shows these 4 layers in more detail.

The *sensor and sensor owners* layer includes physical sensors which can sense an increasingly broad variety of natural phenomena and sensor owners which can be personal, household, private, public, or commercial. Sensor owners have the final say in what data gets to the cloud and who can access the data once it is in the cloud using conditions and restrictions.

The *sensor publishers* (SP) layer manages the detection of online sensors and acts as a middle-man between sensor consumers and sensors and sensor owners. Sensors register with the publisher layer. Sensor data consumers make requests to sensor publishers for specified types of data over specified amounts of time.

The *extended service providers* (ESP) layer builds abstraction on top of sensor publishers. A single ESP can interact with mulitple SPs. ESPs can be used to automatically request data from multiple sensors depending on criteria provided to the ESP. This can be useful if the sensor consumer does not care about the underlying individual sensors but instead queries data at a higher

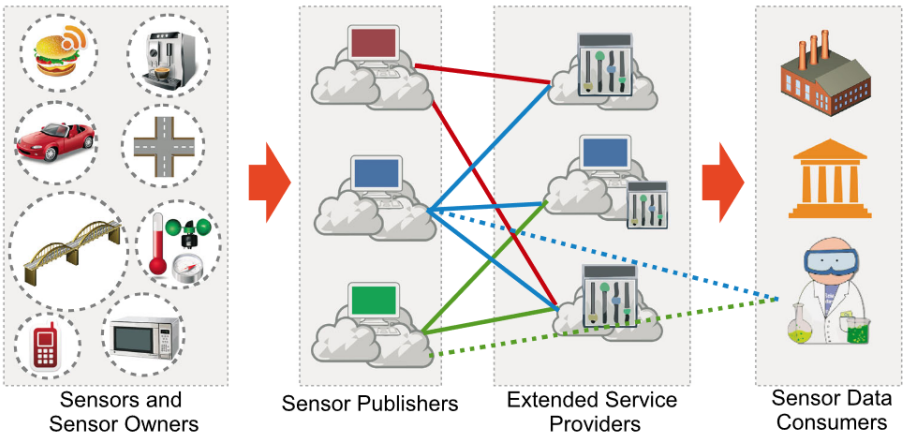

Figure 2: Sensing as a service layers.[48]

level (i.e. all temperature data within a given polygon).

Finally, the *sensor data consumers* layer consist of data consumers who must register with the ESPs and provide valid digital certificates. Consumers can either deal with SPs directly or deal with ESPs. The benefit to dealing with SPs is reduced cost of communications with the ESP. The benefit of dealing with ESPs is higher level querying to data and the ability to query data across multiple SPs.

I find sensing as a service appealing, but lacking in actual implementation details. To Perera's credit, he does mention quite a few open technological challenges that need filled including architectural designs, sensor configuration, sensor management, data fusion, filtering, processing, storage, and energy consumption.

Rao et al.[50] mention several other research challenges for SaaS including the need for a standard distributed computing framework for distributed big sensor data as well as a framework for the real-time monitoring of sensor events.

3.3 Sensor as a Service

In a Sensor as a Service (SenaaS) [7] service model, virtual and physical sensors are combined according to a Service Oriented Architecture. This type of model concerns itself more with the management of distributed sensors than it does with the access, transfer, and governance of data as the sensing as a service model[63]. Figure 3 shows the three layers that make up the architecture in the SenaaS model. Sensors and events can be defined and standardized in XML and other serialization formats such as SensorML[5] and OWL[20].

The *Real-World Access Layer* interfaces to the sensors using adapters which need to be designed for each sensor. Messages from this layer are asynchronously forwarded to the *Semantic Overlay Layer* via callbacks.

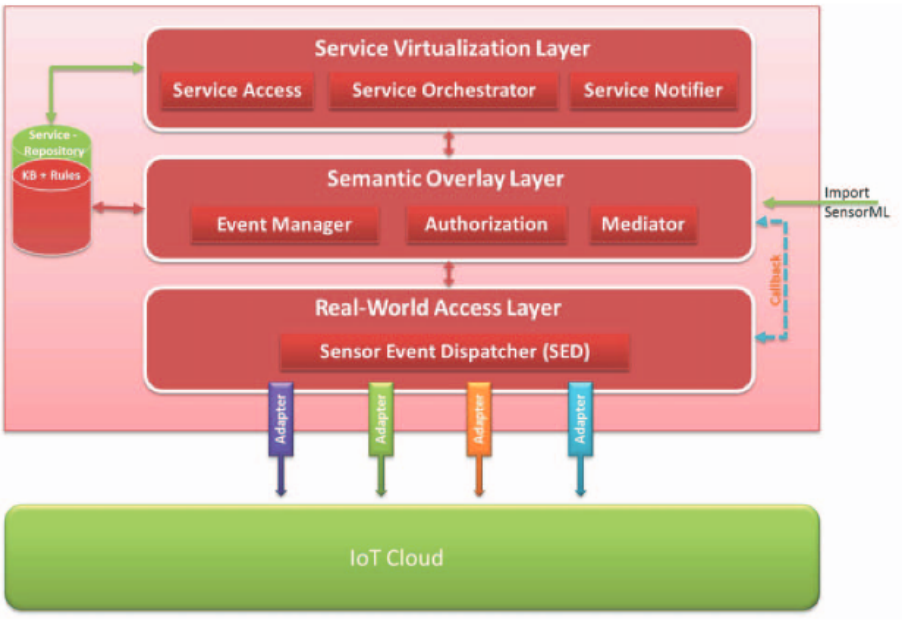

Figure 3: Sensor as a service layers.[7]

The *Semantic Overlay Layer* is responsible for persisting data either in-memory or on disk. This layer also provides for in-memory caching capabilities. Policy based authorization can be implemented in this layer to provide some control over data access.

The *Service Virtualization Layer* provides an abstraction on top of the semantic overlay layer by performing queries based on the access rights of consumers. This layer transforms the results of queries into something that can be consumed by the clients.

3.4 Cloud Deployment Models

NIST[45] provides four types of deployment models in its cloud computing definition: *private cloud*, *community cloud*, *public cloud*, and *hybrid cloud*.

A private cloud is is a cloud where resources are provisioned to a single organization (or multiple parties within a single organization). In this model the organization may deploy their own cloud hardware or use cloud resources provided by a third party.

In a community cloud, resources are provided to a group of organizations that have similar requirements and may be owned and managed by a single organization, multiple organizations, or third parties.

Public clouds provide computing resources to anyone willing to purchase said cloud resources. Public clouds can be owned and manged by anyone,

a government, or any third-party. Generally all hardware in public clouds are managed by the cloud provider.

Hybrid clouds use and provide a combination of the previously mentioned deployment models and can be configured, split-up, and manged in many ways.

Virtual private clouds as described in Botta et al.[11] provide aspects from both public and private clouds using virtual private network (VPN) technology to allow users to manage specific and often times complicated network technologies.

3.5 Mobile Cloud Computing

Mobile cloud computing (MCC) combines mobile computing, cloud computing, and data analytics[61]. Mobile devices such as smartphones make fantastic distributed sensors for temporalspatial data. Not only do they carry a wide array of sensors on-board (microphones, barometers, accelerometers, GPS, compasses, cameras, clocks), but they generally have multiple modes of offloading data (WiFi, bluetooth, cellular, SD cards), and support some pre-processing on the device. Advances and expansion of wireless technologies make it increasingly feasible to connect wireless devices to the network and offload computation and analytics to a scalable and distributed backend[8].

Mobile devices, which are power, compute, and memory constrained can take advantage of uploading data to the cloud for persistence and analysis. Once data is in the cloud, analytics can take advantage of the fact that the cloud can integrate data from many sensors (and other data sources), creating a real-time global view of the network.

Existing MCC application domains include mobile commerce, mobile sensing, mobile banking, crowdsourcing, mobile healthcare, and augmented/virtual reality.

Crowdsourced data capture was an early application of MC[14] and continues to be a main driver of MCC. One example of this is an application, that during amber alerts, allows users to upload pictures to the cloud where thousands of photographs can be analyzed in parallel, helping to track down missing children[53].

Collective sensing and location based services are other major drivers of MCC. Collective sensing takes advantage of the sensors on mobile devices to get a global view of some physical phenomenon. Common sensors that are used in mobile sensing includes microphones, barometers, WiFi chipsets, GPS, and others. Examples of mobile sensing include distributed weather gathering, acoustic classification like Lu et al.'s[40] SouneSense framework which uses distributed Apple iPhones to classify audio events. There has been a surge of research relating to predicting and analyzing real time traffic congestion [58], [30], [32].

3.6 Issues with Cloud Computing

The biggest issues facing cloud computing deal with security and privacy. Subashini et al.[56] go into the specific security and privacy risks associated with the three major cloud deployment models. The following is largely a review of their work.

The deployment model that exhibits the most risks in the PaaS layer since this model requires that consumers manage their own virtual machines, deployment of cloud applications, and configuration. Within this model, the following items are of concern.

Data security is a concern on all deployment models, but especially at the PaaS layer where sensitive data will be stored on remote servers not owned by a client. Since clients manage their own servers in the PaaS layer, they are also required to manage their own data security. Data security issues include cross-site scripting, access control weaknesses, injection attacks, cross-site request forgery, cookie manipulation, hidden field manipulation, insecure storage, and insecure configuration.

Network security becomes an issue when sensitive data is transferred from clients to the cloud backed and visa-versa. If encryption is not used, users could be vulnerable to port scans, ip spoofing, man-in-the-middle attacks, packet sniffing, and more. Even if encryption is used, users can still be vulnerable to packet analysis, insecure SSL configurations, session management weaknesses.

Laws and regulations often require that data not leave or enter certain jurisdictions giving rise to issues of data locality. Users generally do not get to decide where data is stored within a cloud environment as most of those decisions are handled by the cloud provider.

The introduction of distributed systems means that we can no longer make guarantees about data persistence such as ACID (Atomicity, Consistency, Isolation, Durability). Without these guarantees a large amount of data integrity issues surface. We will look at these issues in greater detail when discussion big data persistence models in later chapters.

Data segregation issues occur with multiple virtual resources sharing the same physical resources. Data can be unintentionally leaked or stolen either by attacking the cloud provider multi-tenancy framework or by attacking the virtual servers directly through SQL injection, data validation, and insecure storage.

Large organizations with multiple employees having access to the cloud can create data access issues. Clear policies must be defined, enforced, and updated as to which virtual resources employees have access to.

At some level, consumers are required to trust that the cloud provider they choose will implement security and privacy best practices within their cloud architecture. This does now however resolve the larger issues of security vulnerabilities within cloud software and their communication components.

4 Big Data Persistence Models

Traditional storage methods for meta-data and related products has traditionally made use of the filesystem and relational database systems (RDMS).

Big data by its nature can be structured, unstructured, large, diverse, noisy, etc. Many of the properties of big data do not fit nicely into the structured world of traditional RDMSs.

In-order to meet the needs of big data and distributed sensor networks, we look to the ever growing field of NoSQL (not only SQL) and related Big Data storage models. There are multiple types of data models with different use cases.

According to Song et al.[33] an ideal NoSQL data model strives for "high concurrency, low latency, efficient storage, high scalability, high availability, reduced management and operation costs." The challenges of realizing an ideal NoSQL data model however lie in three main areas[17]: consistency, availability, and partition tolerance.

Several of the persistence models we review do not support ACID (Atomicty, Consistency, Isolation, Durability). A consequence of this is less than perfect consistency. Consistency issues occur when data is stored in a distributed manner with multiple copies. In situations of server failure (or with systems that support different consistency models), situations can arise where multiple copies of the same resource contain different contents.

Vogels and Wener[60] explain the main forms of consistency. Assume a record is being updated across multiple servers. With "strong consistency", any access of that resource after the update will return the updated result. With "weak consistency", subsequent access of that resource is not guaranteed to return the updated result if that access is within a certain "inconsistency window". With eventual consistency, the only guarantee you get is that access to the resource will be show up "eventually" where eventually can depend on many factors.

As the amount of hardware (servers, switches, *etc*) increases in a distributed system so does the amount of hardware errors. Availability refers to the ability to remain operational even as parts of a distributed system drop in and drop out[17]. Gilbert[27] defines availability as "every request received by a non-failing node in the system must result in a response." He goes on further to point out that this definition does allow for unbounded computation since it's possible to wait for a result that never returns.

As the amount of hardware increases in a distributed system, the number of communication packets that drops also increases. The ability to maintain service in the face of some amount of drops refers to partition tolerance[17].

The above ideas are all tied together into the CAP theorem proposed by Brewer[12] which states that in any shared data system, you can only achieve two of the three following properties: Consistency, Availability, or Partition (tolerance). As we review different Big Data architectures, we will examine how they fit into the CAP theorem and what guarantees they provide for these three

major areas of distributed data management.

We also believe, ease of use, maturity of the product, and community (or commercial) support should also factor into the comparisons between data models.

With the above factors in mind, we can begin categorizing and analyzing several major Big Data model solutions.

4.1 Distributed File Systems

Before we look at specific big data storage systems, we will first examine how to store big data using file systems as file systems provide the base storage backend for distributed storage applications. Distributed file systems are not a new idea. Howard et al.[31] describe the scale and performance issues associated with Carnegie Mellon's distributed Unix based *Andrew File System* in 1988.

Generally, distributed file systems were developed as a means of data sharing. Big data however, with the characteristic of large volume, often doesn't fit on a single disk, or a single machine, or a single data center for that matter, required new techniques for storage of very massive data sets. The Google File System (GFS) was one of the first attempts at storing big data using a distributed file system. For that reason, this review will focus on technologies from GFS and onwards.

4.1.1 Google File System (GFS)

Ghemawat et al. at Google introduce *The Google File System*[26] in 2003. Google wanted to design a distributed file system that not only provided scalability, availability, reliability, and performance, which is something that previous distributed file systems could do, but to design it in such a way that it meets the needs of a big data world.

The authors note that in large distributed systems comprised of commodity hardware, failure is the norm rather than the exception. GFS is designed in a way to provide monitoring, error detection, fault tolerance, and automatic recovery of distributed data.

Google noticed that files were becoming much larger and multi-gigabyte sized files were becoming common and should be prioritized. The volume of small files also continued to increase into datasets terabytes in size consisting of billions of small files. GFS examines traditional I/O patterns and block sizes.

The authors stated that there were common read/write patterns among services in their cluster. Reads generally consisted of large streaming reads of small random reads. Writes generally consist of large sequential writes. Once written, files are often not written to again. Small writs are supported, but not prioritized.

GFS prioritizes the ability for multiple clients to append to the same file concurrently in an efficient with minimal synchronization overhead.

Finally, GFS should prioritize high bandwidth over low latency with the assumption that most use cases require processing bulk data at a high rate.

GFS provides a non-POSIX but familiar interface utilizing a directory/file hierarchy and supports creating, deleting, opening, closing, reading, and writing of files.

A GFS system consists of a single master node and multiple chunk servers. Files stored on GFS are divided into fixed-size 64 megabytes chunks and replicated a configurable amount of times on multiple chunk servers. The single master node coordinates locations of chunks in the systems, but only provides routing information. The master node hands off clients to chunk servers to read and write to directly. A chunk block size of 64 MB provides the benefits of less routing requests to the master node and reduces the amount of meta data stored on the master.

The master node keeps track of three types of metadata: file and chunk name spaces, mapping from files to chunks, and locations of chunk replicas. All metadata is stored in memory so that access and manipulation is efficient. The master's metadata transformations are written to an operations log and stored locally as well as remotely and allows for replacing the master in the event of a failure.

The master has several other responsibilities as well. The master is in charge of optimizing replica placement by considering reliability, availability, bandwidth utilization. The master also takes into account the health of the servers that chunks are on including disk utilization. The master can re-replicate chunks when other chunks fail and also rebalance chunks in order to optimize reliability, availability, bandwidth utilization. Files are not immediately deleted. They are marked deleted, but their contents are cleaned up at a later time during a garbage collection phase coordinated by the master.

Similar to other distributed systems, GFS can not make full ACID guarantees and uses a relaxed consistency model. The master node handles namespace mutations (i.e. file creation) and locking so that these operations are atomic. Mutations to files from multiple clients can cause consistency issues when overwriting or inserting into a file. Appends to a file can be made to be atomic if the client does not specify an offset and instead allows GFS to perform the append. Replicas are updated in the same order on all machines so that in general they should all have the same state at the same time. There are instances during failure or when clients cache chunk locations and read from a stale replica where they may retrieve old or inconsistent data.

GFS provides almost instantaneous snapshot capabilities which can be used to create copies, divergent branches of data, or create data checkpoints. Snapshots are created using copy-on-write techniques where if data is not changed, copies can simply point to the original data source, but anytime the original copy changes, a new copy is created. Hence, copies are deferred to first write.

TODO: Find a good way to summarize the results of the paper here.

In an interview with Sean Quinlan[44] (2010), one of the lead architects of GFS, Quinlan discusses how GFS has changed in the 7 years since its

conception.

The biggest change they made to GFS was the creation of a distributed master system rather than a single master. This change came about when data volumes grew initially from tens of terabytes to tens of petabytes in the span of a couple of years. The overhead of meta-data became too great for a single massive master server to handle. Even though clients rarely interact with the master server, something as simple as creating a file could end up in a queue with thousands of other client requests as well. Google decided to build a distributed master server network initially by placing a master server per data center and later multiple master servers per data center. Each master can index 100 million files in memory and there are hundreds of masters.

A major issue that Google ran into internally with GFS was bottle necks due to the volume of small files some projects used. Each file in GFS has meta-data associated with its namespace and the locations of its chunks. The overhead of meta-data caused by many small files was so substantial that Google mandated that applications must find a way to store there information in larger chunks of data and put a quota on the number of files an individual client can create.

As applications changed at Google from batch processing jobs that could take hours to run to a need for more real-time data for user applications, it was clear that Google's previous focus on optimizing for throughput rather than latency needed to change. One solution to this at the GFS level is to leverage parallel writes and perform merges later. In this scenario, if a single write hangs or fails, one of the N parallel writes may succeed. By leveraging parallelism, applications utilizing GFS can give the impression of low latency. Distributed masters also help with latency when performing operations on the file system.

Quinlan concedes that the initial design of relaxed consistency did cause many issues with some of there users down the road and that if he could re-design GFS he push to serialize writes from multiple clients that can ensure replicas remain consistent.

4.1.2 Hadoop Distributed File System (HDFS)

The Hadoop Distributed File System (HDFS), described by Shvachko et al in their 2010 paper, *The Hadoop Distributed File System*[55], is a file system designed for storing big data across a distributed set of servers. HDFS was heavily influenced by GFS and aims to provide an open source alternative to GFS. HDFS acts as the backbone for a myriad of big data persistence and big data analytics frameworks including Spark, MapReduce, HBase, Pig, Hive, ZooKeeper, Avro, Chukwa, and others. A full list of software that integrates with HDFS is maintained at `https://hadoopecosystemtable.github.io/`[3].

As mentioned previously, HDFS is heavily influenced by GFS. Therefor, this review of HDFS will look at the main comparisons between the two distributed file systems, drawing comparisons from Vijayakumari's 2014 comparison paper[59]. The design goals of HDFS and GFS are largely the same.

Support for management of large files as part of large data sets, support for batch computing and big data analytics, and high data availability.

HDFS refers to its meta-data servers as *name nodes* and it data servers as *data nodes* compared to GFS's *master nodes* and *block nodes*.

GFS and HDFS both store data using a hierarchy of files and directories. However, the API to these file systems are not POSIX compliant and require third part APIs for accessing the file systems. HDFS further supports integration with other distributed file systems such as CloudStore or Amazon's Simple Storage Service while GFS is proprietary to Google.

Both GFS and HDFS scale using distributed clusters. Replication of data for providing availability is largely the same.

HDFS uses a permission model that is similar to the POSIX model where files and directories can have separate permissions for owners, groups, and other members. GFS uses a proprietary permission model within its organization that is not based off of POSIX permissions.

GFS and HDFS alike store data in chunk sizes of 64 MB where this value can be configurable. GFS makes use of the Linux kernel's buffer cache to keep frequently accesses data in memory while HDFS uses a combination of a public and private distributed cache. A key difference between the two is that the GFS master node provides clients with the location of blocks where HDFS exposes block locations to allow application to schedule tasks bases on where content is within the distributed system.

GFS uses TCP to communicate between servers and HDFS uses an RPC protocol on top of TCP.

Both of these file systems server similar roles, but HDFS has a much larger impact on the academic realm due the the sheer number of other applications and frameworks that build on top of it.

4.1.3 Haystack

Beaver et al. at Facebook describe a distributed object storage system for storing petabytes of photographs in their 2010 paper *Finding a Needle in Haystack: Facebook's photo storage*[9]. As of 2010, Facebook had stored over 260 billion images.

Haystack was designed to provide high throughput rather low latency and allow Content Delivery Networks (CDNs) to deal with caching and latency issues. Similar to the other distributed file systems, Haystack also wants to provide fault tolerance with the statistical likelihood of software and hardware failures. Another requirement for Haystack is that it is cost-effective along the dimensions of cost per terabyte and normalized read rate. Haystack claims cost per terabyte are 28% less than a similar NFS solution and provides 4x more reads per seconds than a similar NFS based approach.

CDNs are able to cache the most recently accessed photos, but Facebook noticed a pattern where older photos are also accessed frequently. This resulted in a large amount of cache misses on Facebook's CDNs. Cache misses

originally resulted in making requests to NFS shares where each directly contained thousands of photos. The number of seeks required to find a single photo started to become a bottleneck. Taking inspiration from GFS, Facebook created Haystack as a distributed object store focuses on storing and managing its billions of photographs.

Haystack is split into three components. The *Haystack Store*, *Haystack Cache*, and *Haystack Directory*.

The Haystack Store is the main improvement Haystack brings to the table compared to the NFS approach is that Haystack stores individual photos and metadata, called "Needles" into a large (on the order of 100 GB each) continuous files (see figure 4). By using large files instead of many individual files, they can reduce the amount of disk accesses required to find and load individual photographs. 10 TB servers are split into 100 physical volumes of 100 gigabytes each. Each physical volume resides on a single machine, but logical volumes can span multiple machines and multiple physical volumes. When a photo is written to the store, it is stored in a single logical volume, but duplicated multiple times over physical volumes to provide availability. The backing filesystem is XFS which provides small enough blockmaps that they can be stored in physical memory as well as efficient file preallocation. The store also keeps an index file that allows the in-memory mappings to be recreated on failures or server restarts without having to read through the entire file system.

A service called Pitchfork is used to continuously monitor, detect, and repair services in Haystack. In the event of failures, new servers can be brought online and data can be synchronized from the replicas.

The Cache component provides an intermediate layer of photo caching between CDNs and the Haystack Store. Requests for photos contain three URLs. The first URL is a lookup key for the data in the CDN. If the CDN does not have the image cached, the CDN URL is stripped and the photo request is forwarded to the Haystack cache. Again, if the photo is not found in the cache, the Haystack URL is stripped and the request is forwarded to the Haystack Store. These requests are made over HTTP and the format of the requests are as follows: $http : // < CDN > / < Cache > / < MachineId > / < LocaicalVolume, Photo >$.

The Haystack Directory servers provide mappings from logical volumes to physical volumes. They are also in charge of determining whether a read photo should be stored in the Haystack Cache or in a CDN cache. Finally, Haystack Directory servers provide load balancing for writing across logical volumes and reading across physical volumes.

On-top of these three services, Haystack provides a small set of other improvements including compaction which reclaims space of deleted photos and custom binary encoding of indexes to improve space efficiency. In the end, Facebook was able to improve on their photo storage and management by showing performance and efficiency gains over their previous NFS based approach.

TODO: Find a way to summarize

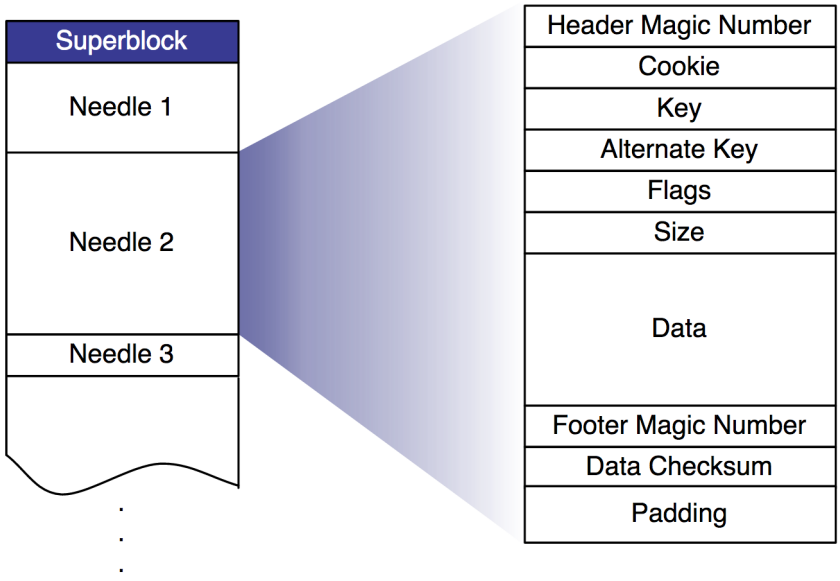

Figure 4: Layout of Haystack Needle within a physical volume.[9]

4.2 Key-Value

The simplest data model for distributed storage is likely the Key-Value (KV) data model [62]. In this model, every piece of data stored is indexed by a unique primary key. Queries to that item all happen via its key to access the value. Values in a KV system can be treated as blobs and the content of the value is irrelevant to the semantics of KV stores. KV systems are popular due to their simplicity and ease of scaling.

Keys in KV systems are the unit parallelism that provide the main means of concurrency. If you want to guarantee transactions, then keys can be naively sharded across servers. This does now however provide safety of data loss in which case a system will strive to provide replication at the cost if ACID compliance. Stores and requests can usually be achieved in $O(1)$ even in distributed systems[49].

If the major advantages are simplicity and query response time[17], the major disadvantage to KV stores is the fact that they lack advanced query capabilities. The only way to query a database is by its unique key. Range based queries, secondary, and tertiary indexes are only supported by a third party systems or application code. Joins can only be performed in application code[1].

Popular KV based solutions include Dynamo, Riak, Voldemort, Redis, MemcacheDB, Ignite and others.

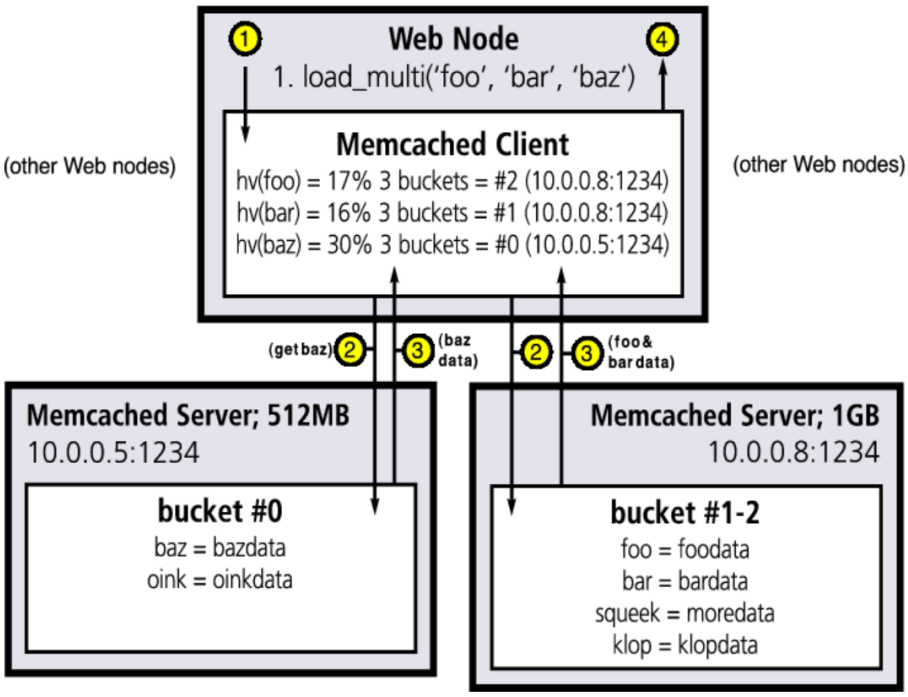

Figure 5: Mecached architecture. [23]

4.2.1 Memcached

One of the earliest examples of a distributed KV store is memcached[23] which was created in part to power the dynamic content of 70 distributed LiveJournal servers. The developer, Fitzpatrick, believed that scaling out on many tiny machines rather than up was the appropriate response for increased data loads. Even though they had SQL database clusters, they could not provide caching on those machines in front of the database due to limitations in address space (32-bit). Fitzpatrick realized that there is a lot of spare memory on the network, even if in small chunks, so the dream of a distributed KV store as a cache was born. Memcached had no persistence guarantees. If a server went down (hardware or software), the data would simply be deleted and any requests to that data would result in a cache miss. Since memcached is used for caching, this is not a huge concern. The real benefit of distribution is that if one server goes down, not all data is lost. It also allows users to take advantage of memory from multiple servers, which wasn't easily possible before.

Memcached is architected as a 2-way hash-table as show in figure 5. The first layer of hashing takes place in the client library. When a client receives a lookup request for a key, that key could live on any of the memcached server instances. The client hashes the key to determine which server the data is

actually on. The request if forwarded to the appropriate server, and the server performs a hash lookup on the provided key and either returns a result or creates a cache miss. This system generalizes to a distributed hash table and provided $O(1)$ lookups and stores. Memory is allocated using slab allocator which used a free list to keep track of free chunks inside of slabs. The protocol allows for fetching multiples keys at one time.

TODO: Review Scaling Memcache at Facebook

4.2.2 Amazon's Dynamo

In 2007, DeCandia and others working at Amazon, released their paper *Dynamo: Amazon's Highly Available Key-Valye Store*[21]. Decandia et al. describe Dynamo as "a highly available key-value storage system that some of Amazon's core services use to provide an 'always-on' experience". Dynamo prioritizes availability over consistency. In order to achieve those goals, Dynamo makes extensive use of object versioning and application-assisted conflict resolution which we will look at in greater detail in the next couple of paragraphs.

As a KV store, Dynamo only provides get and put operations against a unique key and is intended to be used with small values (¡ 1 MB). Amazon, citing experience, mentions that "data stores that provide ACID guarantees tend to have poor availability". Since Amazon's goal for this data store was primarily availability, Amazon decided to use a weaker consistency model which implies that updates will eventually propagate to all peers, at some time in the future. Other guiding requirements for the design of Dynamo include: symmetry where each Dynamo process has the same set of responsibilities as all others, incremental scalability where scaling out has minimal impact on the system or its users, decentralization to avoid single points of failure, and heterogeneity where each process is tuned to the individual hardware of the server it runs on. With these design goals in mind, we next discuss how Dynamo implements these technologies.

Large distributed systems are prone to network, software, and hardware failure. To ensure high availability, Dynamo utilizes optimistic replication, where updates are propagated to N peer nodes in an eventually consistent manner. All nodes in Dynamo's distributed network form a logical ring. Key's are hashed to determine a coordinator node within the ring to initially store the key value pair. The coordinator node stores the key value locally as well as on $N-1$ successor nodes going in a clockwise direction. The list of nodes that end up storing a key are called a preference list. Every node in the ring is capable of determining the preference list for any given key. Dynamo allows for gets and puts on any node in a preference list, skipping over nodes that are unhealthy. Because of this, situations can arise where there are conflicting copies of data when an update hasn't persisted to all nodes in its preference list yet. Dynamo versions all data with a timestamps to allow multiple versions of data to exist in the data store. Using semantic reconciliation, the network is usually able to determine the authoritative version. In certain failure situations where data can

not be semantically reconciled, it is left up to the application to determine which value is correct.

One of Dynamo's stated goals is to provide incremental scalability. When a new server comes online (or others go down), data must be partitioned across the new nodes. Dynamic partitioning is achieved using consistent hashing techniques as described in [36].

Gets and puts are performed in parallel on all N healthy nodes in the preference list. When nodes become unhealthy, a sloppy quorum takes place and the unhealthy nodes are temporarily removed from the ring, and subsequent puts and gets happen on the successive neighbors. To protect against entire data centers going down, replicas span multiple data centers. Dynamo uses Merkle trees to detect inconsistencies between replicas and recover from permanent failures. A gossip-based protocol is used to maintain node membership within the ring.

Dynamo is used to power many of Amazon's backend services and has also been promoted as an offering of Amazon's cloud services.

Riak[37] is an open source implementation of Amazon's Dynamo and was built using DeCandia's paper.

4.2.3 LinkedIn's Voldemort

In 2012, Sumbaly and others from LinkedIn released their paper *Serving Large-scale Batch Computed Data with Project Voldemort*[57]. Voldemort has a similar design to Dynamo in that it utilizes consistent hashing to achieve data partitioning and replication. Clusters form logical rings, and depending on the provided replication factor, consistent hashing is used to select a subset of a cluster to replicate data on.

Unlike Dynamo, Voldemort provides (de)serialization out of the box to/from multiple formats including JSON, strings, Java byte-code, Protobuf, Thrif, Avro, and raw byte streams. Voldemort also supports tuple based compression.

One of LinkedIn's biggest features provides users with a list of people that may know by analyzing social relations between current LinkedIn friends and looking at friends-of-friends. This data is computed with a multitude of graph algorithms such as link prediction or nearest-neighbor calculations. These algorithms run over hundreds of terrebytes of offline data consisting of log files representing social networks, connections, and interactions. This graph structure changes quickly and dynamically meaning these algorithms need to run often to generate and update indexes with valid entries.

Not only does LinkedIn need to handle very large data sets consisting of billions of tuples, they also require the ability to rollback to clean data in the presence of errors. These errors can occur during system upgrade, algorithm changes, incomplete data, or issues with a data source.

Perhaps the biggest improvement Voldemort provides over Dynamo is the ability to compute indexes offline using large distributed computation systems, mainly MapReduce, and supply partitioned data as a result of these computations. This ability allows them to refresh terrabytes of data with "minimum effect

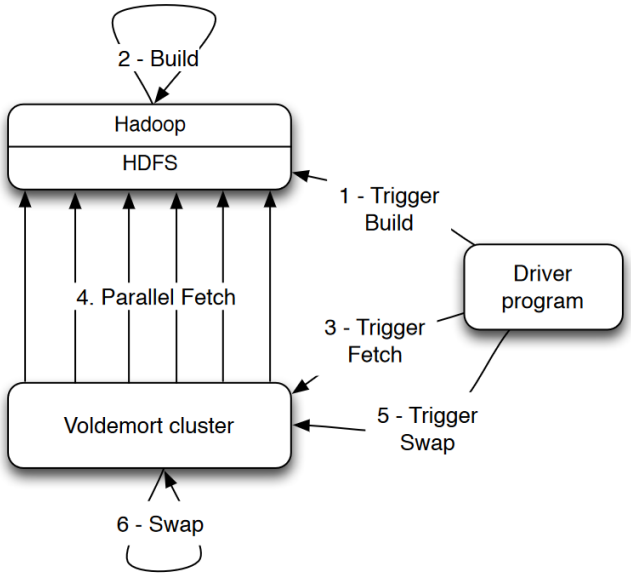

Figure 6: Steps involved in the complete data deployment pipeline. The components involved include Hadoop, HDFS, Voldemort, and a driver program coordinating the full process. The "build" steps work on the the output of the algorithm's job pipeline. [57]

on existing serving latency". Voldemort also takes advantage of utilizing the operating system's page cache for efficient cache management. The collection of these improvements are referred to as *read-only extensions*.

Much of LinkedIn's data can be efficiently computed offline, such as the people you may know feature mentioned earlier. Hadoop provides a map reduce model to distribute graph computations over a cluster. HDFS provides redundancy and availability of data in the face of software or hardware failure. A driver program is in charge of scheduling offline data fetches from Voldemort, map reduce runs, and swapping of live index data back into Voldemort. This process is visualized in figure 6.

Voldemort usages a storage format that memory maps indexes directly into the operating system's address space instead of creating a custom heap based data structure. Voldemort does this to allow the OS to handle caching and page caches which tend to be more efficient than custom data structures. When data is fetched from Voldemort, it is chunked and stored across HDFS in such a way that the file sizes do not become to small which Hadoop does not handle efficiently. Index files contain the upper 8 bytes of the MD5 of the key and then a 4 byte offset to the associated value in the data file. MD5 is used to provide a uniform hash space. Voldemort only uses the top 8 bytes of the MD5 to reduce the overall index size while still providing for a low amount of collisions

($\sim$.0004%). The main advantage of this approach is that it is much more cache friendly. The downside to this approach is that there is increased complexity for dealing with collisions when they do happen.

Voldemort generates its data chunks with a single Hadoop job. The Hadoop job uses number of chunks, cluster topology, store definition, and input location on HDFS. The mapper phase of the Hadoop job partitions data based on the provided routing strategy. A three tuple of generated chunk set id, partition id, and replication factor determine how data is routed to the correct reducer. The reducer phase of the Hadoop job writes data to a single partitioned chunk set. By tuning the number of chunk sets, build phase parallelism can be configured and exploited.

Voldemort makes extensive use of data versioning. Symbolic links point to the most recent version of data on a directory. This feature makes it possible to quickly role back versions of data in failure conditions and also makes it possible to quickly updates indexes from offline data set computations.

LinkedIn continues to use Voldemort to power their people you may know data set as well as their collaborative filtering datasets. They found a 10x improvement in throughput against traditional MySQL solutions at scale. Voldemort also has smaller read latencies than MySQL. The biggest improvement comes from building data sets using MapReduce which scales linearly where the MySQL approach does not scale at all.

4.3 Column

TODO: General introduction of column stores

4.3.1 Google's Bigtable

2008 saw the release of Chang et al's paper *Bigtable: A Distributed Storage System for Structured Data*, which describes a system developed at Google for storing structured data that has the ability to scale to petabytes across thousands of low cost commodity servers and data centers. Bigtable's stated goals are wide applicability, scalability, high performance, and high availability. Bigtable is more complex than simple KV stores and in many ways it can resemble traditional relational database management systems. Similar to the other data stores we've discussed so far, Bigtable's distrubted nature means that it can not guarantee ACID-like transactions and Bigtable relaxes its views on consistency. Bigtable is also unique in that it allows its clients to determine data locality through schema creation and whether data is persisted to memory or backed by disk. These properties allow Bigtable to provide wide applicability to a a very wide range of applications both internal to google and external through Google's cloud computing infrastructure.

Bigtable's structure can be described as "a sparse, distributed, persistent multi-dimensional sorted map" comprised of rows, columns, and column families. Data is indexed ny a combination of row key, column key, and 64-bit

timestamp. Values in Bigtable are simple binary blobs encoded as strings forcing client side applications to make their own serialization and deserialization decisions.

Rows are indexed by strings and reads/writes to individual rows are atomic. Ranges of rows in a table are called *tablets*. Tablets are the main unit of parallelism and load balancing within Bigtable. In this structure, reads to rows can be made more efficient by keeping row ranges small and returning a small number of tablets (which generally come from a small amount of co-located machines). Each row can contain a small number of column families (on the order of hundreds), and each column family can contain arbitrarily unlimited columns. This type of data model flips the traditional row-column model of RDMSs on its head, using columns as the main means of expansion rather than rows.

Column keys are grouped into column families. Data stored in the same column family are generally related. For instance, temperature measurements could be a column family and each column inside the family is an individual measurement. Access control and memory management are performed at the column family level.

Individual stored values can have multiple versions which are sorted in timestamp order which makes tracking of updates possible. Bigtable also provides built-in mechanisms for only storing the last N versions of data or to only keep data received in the last N amount of time.

Bigtable's API allows for storing, querying, and deleting of data. The API allows for meta-data manipulation for access control. It's also possible to run client side scripts on the server to manipulate and filter the data similar to Redis.

Bigtable utilizes Google's distributed file system *Google File System*(GFS) for log and data storage. Data is initially stored in memory, but as it grows, it will be frozen, compacted, optionally compressed, and then written to disk. Once it is on disk, it is immutable and can take advantage of gains in parallelism due to this. Bigtable provides high-availability using a distributed lock service called Chubby[13]. Chubby manages 5 active replicas of which one is elected to serve requests. Chubby manages consistency between replicas using the Paxos algorithm[16], which is an algorithm for forming group consensus in the presence of failure. A master server is used to determine routing to and from tablet servers. However, data does not run through the master server, but directly to the tablet servers once routing has been established. The master server is also responsible for monitoring and managing the health of tablet servers and performing maintenance of creating new tablets in the presence of scale or failure. A B+ tree is used to index into tablets.

Although very successful at Google, Chang does mention that using highly distributed systems provides for many failures including memory and network corruption, unresponsive machines, clock skew, distributed file system quotas, and hardware issues. Next we look at several advances which aim to improve on some of these flaws.

4.4 Document

Document based data models allow data to be stored in structured documents including KV pairs. The underlying document structure can be anything as long as its structure is something that the store can understand. Common document formats include XML, JSON, YAML, BSON, RavenDB, and others[17]. Documents can also store other documents recursively.

Even though documents are restricted to their underlying structure (i.e. JSON, YAML, etc), document databases do not impose a schema like traditional RDMS databases. This allows for painless transitions of data storage when the underlying data change[54]. This is of special concern with heterogeneous sensor networks which can produce large varieties of data in different and changing formats.

Document based data stores often present less of an impedance mismatch between data structures in a programming language and their underlying representation in the data store. Compared to the way data is structured in a RDMS, it's often easier to understand the underlying data structure in a documented based store. There are many common libraries for converting between documents in a document store and a data structure in a particular programming language. Compared to RDBMS, fields in document stores generally do not normalize their data which also enhances the readability of the underlying data structure[49].

An advantage that document based stores have over KV stores is that its possible to create more complex queries. Not only can you query on the primary key, but its possible to create queries over any keys in the document including in sub-documents. Indexes can be created on key and sub-keys. Many document based stores provide range and geospatial based queries. This advantage alone makes document based stores a decent choice for distributed sensor data.

Common document based stores include MongoDB, CouchDB, OrientDB, RavenDB, SimpleDB,

4.5 Graph

TODO.

5 Big Data Analytics

TODO.

6 Conclusion

References

[1] Design.

[2] Fraudpreventiontoolkit.

[3] Hadoop.

[4] IBM. `http://www-01.ibm.com/software/data/bigdata/what-is-big-data.html`.

[5] Sensorml. `http://www.opengeospatial.org/standards/sensorml`.

[6] Eiman Al Nuaimi, Hind Al Neyadi, Nader Mohamed, and Jameela Al-Jaroodi. Applications of big data to smart cities. *Journal of Internet Services and Applications*, 6(1):25, 2015.

[7] Sarfraz Alam, Mohammad MR Chowdhury, and Josef Noll. Senaas: An event-driven sensor virtualization approach for internet of things cloud. In *Networked Embedded Systems for Enterprise Applications (NESEA), 2010 IEEE International Conference on*, pages 1–6. IEEE, 2010.

[8] Marco V. Barbera, Sokol Kosta, Alessandro Mei, and Julinda Stefa. To offload or not to offload? the bandwidth and energy costs of mobile cloud computing. In *INFOCOM, 2013 Proceedings IEEE*, pages 1285–1293. IEEE, 2013.

[9] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, Peter Vajgel, and others. Finding a Needle in Haystack: Facebook's Photo Storage. In *OSDI*, volume 10, pages 1–8, 2010.

[10] Tim Berners-Lee, James Hendler, Ora Lassila, and others. The semantic web. *Scientific american*, 284(5):28–37, 2001.

[11] Alessio Botta, Walter de Donato, Valerio Persico, and Antonio Pescap. Integration of cloud computing and internet of things: A survey. 56:684–700.

[12] Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, 2000.

[13] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350. USENIX Association, 2006.

[14] Andrew T. Campbell, Shane B. Eisenman, Nicholas D. Lane, Emiliano Miluzzo, Ronald A. Peterson, Hong Lu, Xiao Zheng, Mirco Musolesi, Kristf Fodor, and Gahng-Seop Ahn. The rise of people-centric sensing. *IEEE Internet Computing*, 12(4), 2008.

[15] IBM Canada. Smarter healthcare in canada: Redefining value and success. 2012.

[16] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407. ACM, 2007.

[17] Min Chen, Shiwen Mao, and Yunhao Liu. Big data: A survey. 19(2):171–209.

[18] Michael Cox and David Ellsworth. Managing big data for scientific visualization. In *ACM Siggraph*, volume 97, pages 146–162, 1997.

[19] Sanjit Kumar Dash, Subasish Mohapatra, and Prasant Kumar Pattnaik. A survey on applications of wireless sensor network using cloud computing. *International Journal of Computer science & Engineering Technologies (E-ISSN: 2044-6004)*, 1(4):50–55, 2010.

[20] Mike Dean, Guus Schreiber, Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. OWL web ontology language reference. *W3C Recommendation February*, 10, 2004.

[21] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[22] Aurielle Destiche. Fleet tracking devices will be installed in 22,000 ups trucks to cut costs and improve driver efficiency in 2010, 2010.

[23] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124):5, August 2004.

[24] John Gantz and David Reinsel. Extracting value from chaos. *IDC iview*, 1142:1–12, 2011.

[25] Mario Gerla, Eun-Kyu Lee, Giovanni Pau, and Uichin Lee. Internet of vehicles: From intelligent grid to autonomous cars and vehicular clouds. In *Internet of Things (WF-IoT), 2014 IEEE World Forum on*, pages 241–246. IEEE, 2014.

[26] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[27] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[28] Ibrahim Abaker Targio Hashem, Victor Chang, Nor Badrul Anuar, Kayode Adewole, Ibrar Yaqoob, Abdullah Gani, Ejaz Ahmed, and Haruna Chiroma. The role of big data in smart city. *International Journal of Information Management*, 36(5):748–758, October 2016.

[29] Ibrahim Abaker Targio Hashem, Ibrar Yaqoob, Nor Badrul Anuar, Salimah Mokhtar, Abdullah Gani, and Samee Ullah Khan. The rise of big data on cloud computing: Review and open research issues. *Information Systems*, 47:98–115, January 2015.

[30] Ryan Herring, Aude Hofleitner, Saurabh Amin, T. Nasr, A. Khalek, Pieter Abbeel, and Alexandre Bayen. Using mobile phones to forecast arterial traffic through statistical learning. In *89th Transportation Research Board Annual Meeting*, pages 10–2493, 2010.

[31] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, Mahadev Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):51–81, 1988.

[32] Timothy Hunter, Teodor Moldovan, Matei Zaharia, Samy Merzgui, Justin Ma, Michael J. Franklin, Pieter Abbeel, and Alexandre M. Bayen. Scaling the mobile millennium system in the cloud. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 28. ACM, 2011.

[33] A. Jin, C. Cheng, F. Ren, and S. Song. An index model of global subdivision in cloud computing environment. In *2011 19th International Conference on Geoinformatics*, pages 1–5, June 2011.

[34] Guannan Ju, Mengjiao Cheng, Meng Xiao, Jianmei Xu, Kai Pan, Xing Wang, Yajun Zhang, and Feng Shi. Smart transportation between three phases through a stimulus-responsive functionally cooperating device. *Advanced Materials*, 25(21):2915–2919, 2013.

[35] Supun Kamburugamuve, Leif Christiansen, and Geoffrey Fox. A framework for real time processing of sensor data in the cloud. 2015:1–11.

[36] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. ACM, 1997.

[37] Rusty Klophaus. Riak core: Building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10, pages 14:1–14:1, New York, NY, USA, 2010. ACM.

[38] Chun Sing Lai and Malcolm D McCulloch. Big data analytics for smart grid. *Newsletter*, 2015.

[39] Doug Laney. 3d data management: Controlling data volume, velocity and variety. *META Group Research Note*, 6:70, 2001.

[40] Hong Lu, Wei Pan, Nicholas D. Lane, Tanzeem Choudhury, and Andrew T. Campbell. SoundSense: scalable sound sensing for people-centric applications on mobile phones. In *Proceedings of the 7th international conference on Mobile systems, applications, and services*, pages 165–178. ACM, 2009.

[41] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela H Byers. Big data: The next frontier for innovation, competition, and productivity. 2011.

[42] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela Hung Byers. Big data: The next frontier for innovation, competition, and productivity. *Big Data: The Next Frontier for Innovation, Competition and Productivity*, pages 1 – 143, 2011.

[43] Benard Marr. Why only one of the 5 vs of big data really matters, Mar 2015.

[44] Kirk McKusick and Sean Quinlan. GFS: evolution on fast-forward. *Communications of the ACM*, 53(3):42, March 2010.

[45] Peter Mell and Tim Grance. The nist definition of cloud computing. 2011.

[46] NIST Big Data Public Working Group Definitions and Taxonomies Subgroup. NIST Big Data Interoperability Framework: Volume 1, Definitions. Technical Report NIST SP 1500-1, National Institute of Standards and Technology, October 2015. DOI: 10.6028/NIST.SP.1500-1.

[47] President's Council of Advisors on Science and author Technology (U.S.). *Report to the President, big data and privacy : a technology perspective.* Washington, District of Columbia : Executive Office of the President, President's Council of Advisors on Science and Technology, 2014. Includes bibliographical references.

[48] Charith Perera, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. Sensing as a service model for smart cities supported by Internet of Things. *Transactions on Emerging Telecommunications Technologies*, 25(1):81–93, January 2014.

[49] A. Rahien and O. Eini. *RavenDB Mythology Documentation*.

[50] BB Prahlada Rao, Paval Saluia, Neetu Sharma, Ankit Mittal, and Shivay Veer Sharma. Cloud computing for Internet of Things & sensing based applications. In *Sensing Technology (ICST), 2012 Sixth International Conference on*, pages 374–380. IEEE, 2012.

[51] D. Reed, J. R. Larus, and D. Gannon. Imagining the future: Thoughts on computing. *Computer*, 45(1):25–30, Jan 2012.

[52] Nirmalya Roy, Gautham Pallapa, and Sajal K Das. A middleware framework for ambiguous context mediation in smart healthcare application. In *Wireless and Mobile Computing, Networking and Communications, 2007. WiMOB 2007. Third IEEE International Conference on*, pages 72–72. IEEE, 2007.

[53] Mahadev Satyanarayanan. Mobile computing: the next decade. In *Proceedings of the 1st ACM workshop on mobile cloud computing & services: social networks and beyond*, page 5. ACM, 2010.

[54] Sugam Sharma. An Extended Classification and Comparison of NoSQL Big Data Models. *arXiv preprint arXiv:1509.08035*, 2015.

[55] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. IEEE, 2010.

[56] S. Subashini and V. Kavitha. A survey on security issues in service delivery models of cloud computing. *Journal of Network and Computer Applications*, 34(1):1–11, January 2011.

[57] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving large-scale batch computed data with project voldemort. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 18–18, Berkeley, CA, USA, 2012. USENIX Association.

[58] Arvind Thiagarajan, Lenin Ravindranath, Katrina LaCurts, Samuel Madden, Hari Balakrishnan, Sivan Toledo, and Jakob Eriksson. VTrack: accurate, energy-aware road traffic delay estimation using mobile phones. page 85. ACM Press, 2009.

[59] R. Vijayakumari, R. Kirankumar, and K. Gangadhara Rao. Comparative analysis of google file system and hadoop distributed file system. *ICETS-International Journal of Advanced Trends in Computer Science and Engineering*, 3(1):553–558, 2014.

[60] Werner Vogels. Eventually consistent. *Queue*, 6(6):14–19, 2008.

[61] Yating Wang, Ray Chen, and Ding-Chau Wang. A survey of mobile cloud computing applications: perspectives and challenges. *Wireless Personal Communications*, 80(4):1607–1623, 2015.

[62] Silvan Weber. Nosql databases. *University of Applied Sciences HTW Chur, Switzerland*, 2010.

[63] Arkady Zaslavsky, Charith Perera, and Dimitrios Georgakopoulos. Sensing as a service and big data.