CHAPTER

EXPLORING THE EVOLUTION OF BIG DATA TECHNOLOGIES

14

Stephen Bonner, Ibad Kureshi, John Brennan, Georgios Theodoropoulos
Institute of Advanced Research Computing, Durham University, United Kingdom

14.1 INTRODUCTION

Since the adoption of cloud computing, big data has been increasing exponentially in popularity, both computer science and the wider world. This seemingly new paradigm of processing emerges on the heels of e-Commerce and the explosion of Internet-enabled digital devices that allow companies multiple channels and touch points to engage potential customers. The accepted definition of big data is the digital analysis of datasets to extract insights, correlations and causations, and value from data. Different groups have come up with different "Vs" to attempt to formalize the definition of the *big* aspect of this phenomenon. The 3 Vs definition of big data, by Doug Laney for Gartner, states that if it has Volume, Variety and Velocity then the data can be considered *big* [39]. Bernard Marr, in his book Big Data, adds Veracity (or validity) and Value to the original list to create the 5 V's of big data [50]. With Volatility and Variability, Visibility, and Visualization added in some combination to the list by different authors there is now a 7 Vs definition of what constitutes big data [46,3,60,54]. Using sales and advertising as a basis the Vs based definition can be explained as:

- *Volume* – With more data being collected, logged, and generated organizations need larger storage to retain this information and larger compute to process it.
- *Velocity* – Through online transactions and interactions the rate at which the Volume is being created vastly exceeds data generated from person-to-person interactions. Online systems are also expected to react in a timely manner meaning that the data needs to be processed as quickly as it gets ingested.
- *Variety* – A digital transaction gives more than just a sale, even a customer browsing certain sections of an online store is valuable information, whether or not a sale is made. In an online transaction, person A buying Object-X is not the only information that can be extracted. Socio-economic, demographic, and consumer journey information can all be collected to improve future sales and advertising. The problem becomes more complex within the inclusion of data from traditional and social media.
- *Veracity* – Large volumes of disparate data being ingested at high speed are only useful if the information is correct. Incorrectly indexed data or spelling mistakes could make complete datasets useless and thus the veracity is important.
- *Validity* – Inferences derived from the data may also not always be accurate. As the saying goes correlation does not always imply causation and so the validity of the insights derived from the data need to be validated.

Software Architecture for Big Data and the Cloud. DOI: 10.1016/B978-0-12-805467-3.00014-4

- *Value and Volatility* – Value is a multifaceted property of big data. As the volume of data grows the incremental value of each data point begins to decrease. As the variety of data available increases, not all the data may aid in product development, sales, or system management. Data may also lose relevance over time. In this case with the dataset not being a part of the business process how are the costs of retention justified. Big data is not the retention of all data; some data needs to remain volatile.
- *Value and Visibility* – The key motivation behind big data is to extract *value* from the data. Many would argue that this involves extracting hidden meaning from the available data. With valid and valuable data information that is not visible may be extracted to make more informed business decisions.
- *Variability* – An extension of variety, variability exists where a metric of a dataset being collected changes in an unpredictable manner that affects the entire business processes. Unexpected purchasing behavior due to factors external to the supply chain means retailers and vendors would need to dynamically tailor their pricing and advertising strategies accordingly.
- *Visualization* – The adoption of big data within corporate business processes means the data needs to be accessible to all who need it. Large dumps of statistical data may not always be easily interpretable at a glance. Visualizing the data and any inferences is an additional important facet of big data.

14.2 BIG DATA IN OUR DAILY LIVES

Michael Cox and David Ellsworth in their 1997 paper titled "Application-controlled demand paging for out-of-core visualization" used the term big data to refer to the phenomenon of datasets not fitting in memory or on local disk especially in the context of visualization [18]. This is the first use of the term big data in published literature at the time. However, the concept of the data explosion goes back many decades [58]. Within scientific endeavors the Large Hadron Collider program at CERN generated 30 petabytes of data in Phase 1 and is expected to generate approximately 25 gigabytes per second when all 4 experiments are operational in Phase 2 [43]. The Large Synoptic Survey Telescope will generate 15 terabytes of data a night once operational in 2019 [47].

Big data attracted the attention of the wider world through two main seminal developments that came in the form of the Google File System in 2003 [24] and MapReduce in 2004 [20]. The implications of sifting through a multitude of logs, that had hitherto only been methods to debug errors in the system, to profile visitors and target advertising, brought data-centric computing in vogue. Webservers, with their logs and browser cookies, allow organizations to collect detailed information about visitors that includes, but is not restricted to, their device, location, time spent on web pages, journey to the webpage, and the journey beyond. Processing this information through data mining and analytics the visitor's age and gender can be inferred, if not already provided through social media or other account information. Over time the data collected also leads to socio-economic and demographic inferences. All of this helps services, the likes of Google or Facebook, to perfectly target advertising to its visitors, making their platform the perfect channel for other companies to engage their target market.

At its core, the data mining taking place aims to create a profile of the person being observed. The person-centric model being created includes their socio-economic and demographic information, their tastes, and their behavior. Plenty has been written about the use of big data for advertising and so the

focus here will be to use the same principles to create a person-centric model that can be applied to new developing paradigms such as Smart Homes. With the advent of the Internet of Things revolution, more metrics can be collected to create holistic models of human behavior. While currently the focus is centered on making devices *Internet enabled*, protocols and encryption techniques are being developed to capture and transport the data deluge to be computed, the *killer apps* will come when this data is processed.

In a futuristic smart home, the agent managing the household will need to make decisions regarding scheduling, brokering, and operation. Here is an example of a hypothetical scenario to understand the decisions the smart home will need to make and the data that is available to it. It is 4 a.m. on a Tuesday in March; the occupants have set their alarms to wake up for 6 a.m. One has a meeting at 9 a.m. with, according to the GPS software, a 1-hour commute beforehand. Several other meeting are scheduled which are geographically dispersed. The other homeowner has nothing scheduled until lunchtime but usually leaves for work at 8:45 a.m. Both have the same entry in their calendars for 6 to 8 p.m. The dishwasher is almost full. The washing machine is full. The weather is going to be clear and sunny, with a 5 mph breeze. Sunrise is at 7:20 a.m. and sunset is at 6:08 p.m. The maximum temperature during the day will be 14°C with a minimum of 9°C during the night. The calendar shows that it will be another early start on Wednesday morning.

The agent can use all this information plus historic information and make the following decisions. The coffee maker with Owner A's coffee preference needs to be triggered at 7:45 and then again at 8:30 with Owner B's preference. After 9 a.m. the dishwasher will have the breakfast crockery and utensils and will be sufficiently filled for a full run. The washing machine too will need to run. When it returns at least one of the two electric cars will need a full recharge if not both. Home lighting and heating will not be required till after 8 p.m. as both owners will be out till then and in case they are present before 6 p.m. it will still be daylight. Due to the early start on Wednesday the homeowners are likely to sleep before 11 p.m. implying, at a maximum, a 3-hour user driven power load. A typical Tuesday night load can be extracted from historical records that include climate control information. Based on previous behavior the home agent can infer that if both homeowners are scheduled to stay out till 8 p.m. then they will eat out and so the home stove will not need to come on. Due to the low wind speeds the wind turbine in the garden is not likely to generate much power; however, the solar panels will be able to generate a predictable amount. The CCTV shows that the homeowners return at 10 p.m. and are not accompanied with any other guests so no changes to the earlier model are needed.

Based on all these metrics and inferences the home can automate most of its processes. Further it can predict how much power load it needs to deliver through the day and what the size of the shortfall will be (if any). The home can negotiate with the power grid and get preferential rates for its deficit requirements. If better rates are given during the day, then the home appliances can be scheduled to run during those times. This saves the solar charged battery power for the time the homeowners are back in or to charge the car.

While this may sound like a utopian Jetsons fantasy, big data, cloud computing, and Internet of things have enabled considerable research and development in this area to make it a reality [35,74,76]. Using the calendar, previous repeated behavioral patterns, data from various Internet enabled devices, and other external sources (e.g., weather services) a smart home can build a complete profile of its occupants, their preferences and their expected behavior. Similar to the model created through people's online behavior that is used in advertising, these highly detailed models are only achievably by processing large quantities of unrelated datasets that need to be sourced from disparate sources. The

various devices in the hypothetical scenario continuously generate data that needs to be harvested and used in near real time to account for any changes. This is big data in the home.

14.3 DATA INTENSIVE COMPUTING

The term data intensive computing encapsulates the technology designed to store, manage and process big data [34]. Data intensive computing contains two key areas, applications and frameworks, both of which exploit data parallelism. Data parallelism is the notion that data is distributed among nodes in a parallel computer and therefore can be processed in parallel. It has been argued that data, as opposed to task parallelism, is often the easiest way to create a parallel program [33]. Data intensive frameworks are specialist software, designed to create and run data intensive applications. While data intensive applications are usually data parallel programs, whose main function is the manipulation of massive datasets.

14.3.1 BIG COMPUTE VERSUS BIG DATA

Simply processing large datasets is typically not considered to be big data. Groups like *Conseil Européen pour la Recherche Nucléaire* (CERN) and *Transnational Research In Oncology* (TRIO) have been using High Performance Computing systems and scalable software to analyze very large datasets. However, this is considered *compute-centric processing*. Typically, a mathematical algorithm is used to generate results from the data that forms the input. This is true for computational fluid dynamics, image processing, and traditional genome analysis.

From a components perspective, little differentiates a big data machine from a supercomputer; however, the philosophy of design and interaction of components places these systems in two classes. Table 14.1 outlines the salient differences between big compute and big data systems.

Computers, large and small, are very efficient and quick at processing decimal numbers and mathematical equations. While not always true, typically compute-centric processing takes as input small quantities of data and generates large quantities of data while processing or as output. As the power and thermal limits of silicon were approached, multiple processors were ganged together to parallelize and speed up processing. Compute-centric applications typically parallelize the processing by distributing the data and the instruction to multiple processors that can be as close as being located on the same chip, up to being geographically remote and only accessible over the Internet.

Taking traditional high performance computers (HPC) as an exemplar system, processing elements (PE) are typically located across a tightly coupled local area network operating at network speeds between 1 to 100 Gbps. The biggest bottleneck to a compute-centric parallel program on an HPC system is the network. Complex programs are not able to scale to bigger systems due to large global communication operations. Input data therefore cannot be distributed in the same manner as the instruction set, from one system across the entire parallel machine. A dedicate fast shared storage device is used to make the data available to all processing cores. As all the data is clumped together, even if the data domain is partitioned within the algorithm, each PE still needs to traverse the entire dataset to find the required subset. First and foremost, this is a waste of compute cycles, and secondly it causes unsus-

Table 14.1 Big compute systems vs big data systems

Component	Big compute systems	Big data systems
Data	Centralized shared storage	Local storage managed centrally
Network	High bandwidth and low latency required for scalability	Latency plays minimal role, most bandwidth required during data loading and unloading phases
System Design	Tightly coupled systems	Run on almost anything approach – PEs can be heterogeneous with consumer grade networks
Task Management	Task concurrency required. System can be divided among jobs but each job must run as a single unit	Task and data replication allows for out of order execution of tasks and jobs can also be split
Job Resilience	If a PE involved in a job fails, the job fails and needs to be restarted	Replication of tasks allows for the failure of PE without affecting the job
System Resilience	Systems can cope with loss of PE. Single points of failure: Centralized Storage, Controller Node (redundancy possible), Internal Networking	Built-in redundancy in nearly all components. Default operation includes replication. Single Point of Failure: Internal Networking
Programming Model	Message Passing Interface (MPI), Symmetric multiprocessing (SMP), Parallel Virtual Machine (PVM)	MapReduce, Bulk Synchronous Parallel (BSP)

tainable loads on the storage device (e.g., 50–100 PEs synchronously accessing 1 storage server to find the relevant 1 GB of data to process). For example, if we consider a CFD job with a detailed model ($>$1 GB) of the object then each PE will request that file and then only load into memory the *chunk* of the model it must process. When designing a typical HPC system (for CFD or FEA workloads) the key metrics in order of importance is processor clock speed, processor density, network interconnect, processing power and network capability of centralized storage, and finally memory per PE.

The main distinction of the big data computing paradigm is that the processing algorithms and systems are usually designed with data centricity in mind. Data centric systems devote the majority of the running time to performing data manipulation and I/O over numerical operations. The optimization of algorithm is secondary to the data management. Within the MapReduce ecosystem (discussed further down) data locality is a major component to the concept of data centricity. Unlike the HPC systems above the data is not stored at a single point. At ingestion time it is in fact divided and replicated across a distributed compute system, and stored local to the processing elements. That way each PE only needs to deal with its local subset of data without needing to search for it. The PEs also do not need to compete with each other for bandwidth over the network. This creative distribution of data ensures that processing instructions only go to the PEs that have direct access to the relevant data. The replication of the data subsets ensures further and much improved parallelization. If two instructions require access to the same subset of data, the replication ensures that two separate PEs can run both instructions simultaneously without any contention for network or disk. When designing a big data system, the metrics to keep in mind are size of PE level storage, speed of PE level storage, memory per PE, processor clock speed, and network interconnect. This is another reason for the penetration of big data within the commercial sector. HPC systems have large power overheads – leading to high processor density, and require highly specialized networking components to deliver performance. Conversely

big data systems can be very efficient using high quality workstations loosely coupled with commodity networking equipment. This also makes big data systems easy to deliver over cloud computing infrastructure, lowering the financial barrier to entry.

14.3.2 DATA INTENSIVE APPLICATIONS

There are a variety of computation types that can be considered data intensive. Such applications often use massive quantities of input data to derive some important value from it. Indeed, increasingly a company's success is driven by their ability analyze and draw conclusions from enormous quantities of disperse data sources. The need to derive new information from data has led to the emergence of several sub categories of data intensive applications including:

- *Data Mining* – Data-mining algorithms are a broad class algorithms used to extract certain key features or metrics from a given data. The types of algorithms used and features extracted from data vary upon the domain the data belongs to. The algorithms used to mine astronomy data, for example, would differ from those used to mine financial market data [16].
- *Machine Learning* – Machine learning algorithms are a varied set used broadly to study and learn from datasets with goals including pattern recognition and future prediction. Machine learning is closely related to the data-mining field and recent improvements in algorithms and the power of modern hardware have accelerated progress in the field. The use of larger and more varied training data sets have increased accuracy of any predictions created [42].
- *Real Time Processing* – The introduction of modern software and hardware has increasingly made the real time processing of massive datasets a reality. Such real time analysis is often named stream processing [42]. In stream processing model, data is processed, queried and analyzed immediately as it arrives into the system. Increasingly a data stream from a generation source is sent directly to a compute resource for processing before then being archived for latter access. This is in contrast to the batch processing method, where data is first stored and processed in bulk at a later date.
- *Graph Processing* – Many domains can naturally be represented as graphs as they can capture the inherent interconnecting nature of real world phenomena [8]. Such domains include social networks, semantic web, protein networks and computer networks among others. Due to this, there have emerged a class of algorithms designed specifically to extract metrics and topological features from graph objects, giving valuable and unique insight into the dataset.

14.3.3 DATA INTENSIVE FRAMEWORKS

Data intensive frameworks are a class of software designed to enable the efficient creation and running of data intensive applications. Data intensive applications pose interesting and unique demands on the underlying hardware as data transfer, not processor speeds, limits their performance. As such, data intensive frameworks make important considerations and compromises to optimize for data processing in their architecture design and implementation [68]. The need to process massive quantities of data being generated called for a paradigm shift in the mind-set of system and application designers. This shift called for systems that excel at ingesting, moving, manipulating, and retrieving data on an unprecedented scale. This poses real challenges for both hardware and software. Many of the resulting data intensive frameworks emerged from Internet-focused companies and research institutes,

for example MapReduce from Google and Dryad from Microsoft. Data intensive applications prioritize input/output (IO) operations, specifically disk and memory access, over CPU based computation [66]. Both compute and data intensive computing are performed of distributed clusters, usually with a shared-nothing architecture. Although the famous Moore's Law has observed that CPU performance has doubled, due to progress made in the miniaturization of transistors, approximately every two years, IO has experienced a much slower increase in performance. This has led to the notion that compute is considered cheap and data IO is expensive. As data intensive computing is usually performance upon a distributed system, the bandwidth and latency of the network are also an important factor in performance, as large quantities of data needs to be transferred across it. However, the movement of massive datasets across the network has proven to be exceptionally time consuming. For example, a system with a 10 GB network running at 80% utilization, would take approximately 11 days to transfer a dataset 1 petabyte in size [37].

Due to these demands, data intensive frameworks are designed to provide such optimizations as data locality and fault tolerance to optimize the available hardware. There has been a general shift in the management of the complexities introduced by fault tolerance and data locality from hardware to software in the form of data intensive frameworks. In addition, data intensive frameworks mask the complexity of the underlying hardware from the application developer and automatically handle data partitioning, scheduling, and parallelization [33].

14.3.4 MAPREDUCE AND GFS

In 2003 and 2004 Google introduced two key concepts to the research community; both would become some of the cornerstones of the data Intensive computing research landscape. Firstly, Google introduced the concept of MapReduce, which it had developed internally as a conceptually simple, yet extremely powerful new programming paradigm for processing massive datasets in parallel. The second paper introduced was the Google File System (GFS), a distributed and fault resilient file system. The GFS system allows for large files to be split into smaller blocks, which can then be distributed across all nodes in a cluster. These file blocks allow for easy resilience against hardware failure, as each block can be replicated across a range of separate machines. The following sections will give an overview of the design and functionality of both GFS and MapReduce and explain why this model has become so successful for Data Intensive Computing.

14.3.4.1 The Google File System (GFS)

In 2003 Google introduced the distributed and fault tolerant GFS [24]. The GFS was designed to meet many of the same goals as preexisting distributed file systems including scalability, performance, reliability, and robustness. However, Google also designed GFS to meet some specific goals driven by some key observations of their workload. Firstly, Google experienced regular failures of its cluster machines; therefore, a distributed file-system must be extremely fault tolerant and have some form of automatic fault recovery. Secondly, multigigabyte files are common so I/O and file block size must be designed appropriately. Thirdly, the majority of files are appended to, rather than having existing content overwritten or changed, this means optimizations should be focused on appending files. Lastly, the computation engine should be designed and colocated with the distributed file system for best performance [24].

With these goals in mind, Google designed the GFS to partition all input data in 64 MB chunks [24]. This partitioning process helps GFS achieve many of its stated goals. As such, the comparatively large size for the chunks was not chosen by chance. The larger chunk sizes result in several advantages including less metadata, a reduction in the number of open TCP connections and a decrease in lookups. The main disadvantage to this approach is that space on the distributed file system could be wasted if files smaller than the chunk sizes are stored, although Google argues that this is almost never the case [24]. In order to achieve fault tolerance, the chunks of data are replicated to some configurable number of nodes; by default, this value is set at three. If the cluster comprises a sufficient number of nodes, each chunk will be replicated twice in the same rack, with a third being stored in a second rack. If changes are made to a single chunk, the changes are automatically replicated to all the mirrored copies.

From the point of view of the architecture, GFS is conceptually simple, with cluster nodes playing only one of two roles. Firstly, nodes can be data nodes, whose role is to physically store the data chunks on company's local storage and comprise the vast majority of all the cluster nodes. The second class of node is the master node, which stores the metadata for the distributed file system including the equivalent of a partition table, recording upon which nodes chunks are stored and which chunks contain certain files. The GFS has just one master node per cluster. This enables the master node to have a complete view of file system and make sophisticated data placement and partitioning strategies [24]. The master node also ensures that if a data node goes down, the blocks contained on that node are replicated to other nodes, ensuring the block replication is maintained. The one obvious problem with the single master strategy is that it then becomes the single point of failure for the cluster, which seams counterintuitive considering one of the main goals of GFS was resilience against hardware failure. However, the current state of the master node is constantly recorded, so when any failure occurs, another node can take its place instantly [24].

14.3.4.2 MapReduce

MapReduce is both a powerful programming paradigm and a distributed data processing engine, designed to run on large clusters comprised of commodity hardware originally introduced by Google via a 2004 paper [20]. MapReduce was specifically designed as a new way of processing the massive quantities of data required by a company like Google. Its programming model takes inspiration from functional programming and allows users to easily create scalable data parallel applications, whilst the processing engine ensures fault tolerance, data locality and scheduling automatically. MapReduce is not designed as a replacement for traditional parallel processing frameworks such as MPI; instead it is a response to the new class of applications demanded by the big data phenomenon. When Google originally designed the MapReduce system, the following assumptions and principals guided its development [64]:

- MapReduce was designed to be deployed on low-cost and unreliable commodity hardware.
- This hardware was loosely coupled and configured as a Redundant Array of Independent Nodes (RAIN).
- Nodes from the RAIN were assumed to fail and thus could be removed at any time. These failures should have no impact on any running jobs or result in any data loss.
- The MapReduce paradigm was designed to be highly parallel, yet abstract enough to allow for fast and easy algorithm development.

In MapReduce, the compute engine and the distributed file-system are designed together and are tightly coupled. The system utilizes this tightly coupled nature to create the key performance driver of MapReduce – data locality. Data locality ensures that the required computation is moved to the data as the node that holds the data will process it [27]. This is an advantage in a modern compute cluster environment, as data transfer is often the bottleneck in application performance and bringing the compute to the data will remove the need for a costly network transfer.

From a conceptual point of view, MapReduce can be considered as just two distinct phases: Map and Reduce [44]. In order to achieve some of its fault tolerance and scalability goals, the MapReduce system places some limitations on the way end users create their applications. Perhaps the most challenging, from an end user's perspective, is that Map tasks must be written in such a way that they can operate completely independently, and in isolation, on a single chunk of the overall larger dataset. Any operations that require some form of communication must be performed in the Reduce phase, which can aggregate the required result from data passed to it by a series of Mappers. To create a MapReduce application an end user must be able to express the logic required by their algorithms in these two phases, although chaining multiple MapReduce iterations together can accommodate more complicated tasks. It is also possible to create Map only jobs for tasks that do not require any sort of accumulations, such as some data cleaning or validation tasks. From an end user's perspective, one of the key strengths of the MapReduce paradigm is that their applications are completely removed from the often-challenging tasks usually associated with parallel computing. The backend system of MapReduce handles the data distribution, fault tolerance and scheduling for the end user's automatically [64]. This frees users to just focus upon the creation of new algorithms and the parallelization is handled automatically. This lowers the complexity of writing algorithms massively and helps democratize the creation of parallel programs so nonspecialists can harness the power of modern compute clusters [20].

The MapReduce system uses key/value pairs as the input and output for both of the stages. The input data is presented to the Map function as key/value pairs and, after processing, the output is stored as another set of key/value pairs. In between the Map and Reduce phases, common keys in the Map output are grouped together so all the associated values are available for processing in the same Reduce task. The final processing and result from the Reduce task are again output as key/value pairs [20]. One of the key performance drivers of MapReduce is that the Map phase is highly parallel. By default, if the input data resides in m blocks, then m Map tasks will be spawned. As GFS ensures that blocks are distributed across the entire cluster, the Map tasks will be executed on many nodes simultaneously. This is the simplistic model of MapReduce and gives a good representation of how data will flow through an application but it does not discuss some key behind the scenes operations performed by the system.

Greater insight into the operation and intricacies of MapReduce can be gained through the analysis of the word count application, also known as the hello world of MapReduce. For this example application, the input to the program will be a collection of text documents stored on a GFS-like file system and will be completed in a single MapReduce phase. The collection of documents would be split into m 64 MB chunks automatically by the GFS. Once the MapReduce program was launched m Map tasks would be created, wherever possible, upon the nodes containing the relevant file chunks. In the word count application, the role of the Map task is to split the text data contained in the block, using whitespace, into a sequence of individual words. The Map function would then emit its series of

intermediate key/value pair with each word located being the key and the value being an integer value of one. Pseudocode representing this process is shown as Code 1.

```
Map:
     void Map(string document) {
           for each word w in document {
                 Emit_Intermediate(w, "1");
           }}
Reduce:
     void Reduce (string word, list<string> values) {
           int count = 0;
           for each v in values {
                   count += StringToInt(v);}
           Emit_Final(word, count);
     }
```

CODE 1

Pseudocode for a MapReduce version of word count

An example output from the Map phase would be $(w1, 1)$, $(w2, 1)$, ..., $(wn, 1)$. The transfer of data between the Map and Reduce phases is handled by a process called shuffle and sort. The role of the shuffle and sort phase is to collect and sort the values associated with a specific key so that they are all presented to a single Reduce task. This phase can be a performance bottleneck, as all the intermediate data from the Mappers is first written back to disk before then being transferred over the network to the nodes that will run the Reduce task. In the word count application an example input to the Reduce function would be $(w1, [11111])$, for a word that had five instances in the original input file. The Reduce function would then simply sum the integer values for all keys and emit the total, along with the original word as the final output of the application – $(w1, 5)$.

Whilst MapReduce has proven to be extremely powerful and popular, it is not without fault and has received some criticism within academic literature [21,56]. The main arguments against MapReduce centers around a few key areas including the comparatively reduced functionality, its lack of suitability for certain computation tasks, a still relatively low-level API and its need for a dedicated cluster resource. When contrasted with other data management and query systems, such as SQL, MapReduce can appear to offer limited functionally. Simple standard relation database operations such as joins are complicated in MapReduce and often require sophisticated solutions. Although several strategies, including Map-side, Reduce-side and Cascade joins, have emerged to enable the functionality, the framework was clearly not designed with workflows involving numerous complicated joins in mind [1]. MapReduce completely removes the data schema used by traditional databases so all data is stored without structure or an index, meaning that it is unable to utilize the possible optimizations offered by structured and semistructured data [21]. The lack of an index means that the entire dataset must be traversed to search for a specific portion of the data, which can be costly, especially with massive datasets. In its original incarnation there is no higher-level language for MapReduce, and users must write their applications using the still low-level API. When compared with writing SQL queries, for example, the MapReduce API has a greater level of complexity and requires more lines of code. In addition, MapReduce code is often less portable and tends to be very data-specific [44]. For certain data

processing tasks, particularly those that require many iterations over the same dataset, the MapReduce paradigm is unsuitable. Unfortunately, many state-of-the-art machine learning and graph processing algorithms display exactly these very characteristics [78]. From a system point of view, MapReduce is often deployed on its own dedicated hardware, as the system does not lend itself to resource sharing with competing frameworks such as MPI. This can increase costs for an organization as it potentially must purchase and maintain two clusters if the requirement for both systems is present within the organization.

Despite these limitations, MapReduce has proved to be extremely popular in both industry and academia. Although it should be clear now that MapReduce is best suited for use in a specific class of application, when a massive amount of data needs processing and when the required processing fits well within the data parallel model. The original implementation of MapReduce cannot be, nor was it designed to be, a replacement for parallel processing engines such as MPI or structured database systems such as SQL. Instead it compliments these systems and fills the void when the required workload doesn't fit into either paradigm.

14.4 APACHE HADOOP

14.4.1 HADOOP V1

Apache Hadoop is an open-source project first developed by researchers at Yahoo [4]. It was designed to replicate the functionality of both Google's GFS and MapReduce system. As such, Hadoop originally launched with two components; the Hadoop Distributed File System (HDFS), designed to replicate GFS, and its own implementation of the MapReduce runtime and programming model [67]. Since the code was first made publicly available in 2007, Hadoop has grown to become the de facto standard for big data processing. Since it is a replication of the original Google ideas, Hadoop inherits the same set of strengths and weakness as was previously discussed. The only major differences between them are that Hadoop is written in the Java programming language, compared with the C++ of the original Google system, and that Hadoop is open-source, thus it can be used and modified free of charge.

Hadoop is designed to run on a distributed shared nothing cluster of commodity machines. The original Hadoop implementation runs as a series of daemon processes, with nodes in the cluster taking one of two roles, master or slave. Each of the four separate daemon processes runs within a Java Virtual Machine (JVM). These daemon processes (shown in Fig. 14.1) are:

- *JobTracker* – Master process for the MapReduce component that controls the submission and scheduling of jobs on the cluster. Runs on the master node.
- *NameNode* – Master process for the HDFS that keeps track of how data has been distributed across the available DataNode. Runs on the master node.
- *TaskTracker* – These processes are the worker components of the MapReduce system. The TaskTracker demons themselves do not perform the computation; instead they control the spawning of separate JVMs for each MapReduce job. These are run upon the slave nodes.
- *DataNode* – These processes control the data stored in the HDFS. These are also run upon the slave nodes.

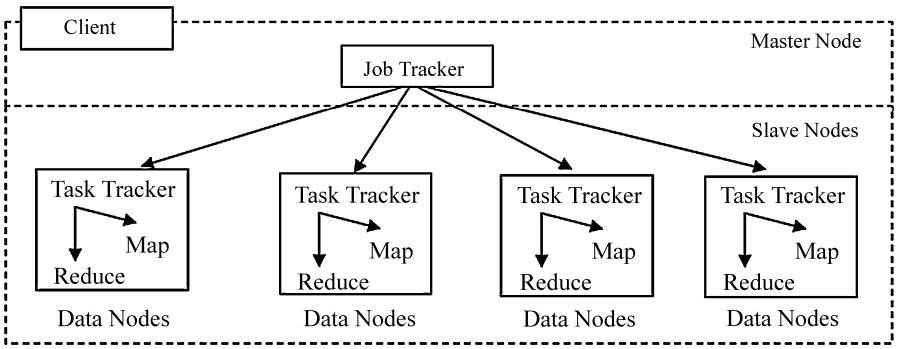

FIGURE 14.1

Interaction between various Hadoop processes [71]

14.4.1.1 The wider ecosystem

Due to the popularity of Hadoop, various research teams have greatly extended its functionality past its original design brief. Many of these projects have been incorporated under the Apache Hadoop banner. These technologies include: HBase, Cassandra, Hive, Pig, Impala, Storm, Giraph, Mahout, and Tez. These systems are designed to introduce additional computing paradigms into the Hadoop ecosystem. Some of the most popular are explored below:

- *HBase* – Apache HBase is an open-source implementation of Google's BigTable system [15]. HBase provides a NoSQL based, fault tolerant, distributed and scalable database layer that resides on top of the HDFS [12]. HBase uses the wide column model from BigTable to store the data, as such it has no schema. Unlike HDFS alone, HBase enables real-time random read and write access to the data contained within a table. HBase provides no traditional query language, so users must access the data via a standard MapReduce job [12].
- *Cassandra* – Apache Cassandra is another database solution for Hadoop that is highly available and scalable, started in 2010 as Facebook's alternative to BigTable [38]. As such Cassandra is similar in functionally to HBase as it is a distributed NoSQL database that can provide high availability, linear scalability and data dissemination across multiple geo-distributed data centers. However, Cassandra supports the Cassandra Query Language (CQL) to provide a way to query data using a limited, SQL-like language [14].
- *Hive* – Apache Hive is a project created by Facebook to provide not only a structured data store, but also a way of querying the data via an SQL-like language called HiveQL [69]. Hive incorporates a query created using HiveQL that is automatically translated into the required number of MapReduce tasks, enabling users who are familiar with SQL to easily start processing data via Hadoop. A key element is that Hive will automatically perform any requested joins [69].
- *Pig* – Apache Pig was created to be a high-level data-flow language, comprising two elements: the Pig data-flow language named Pig-Latin and the Pig execution environment which enables the job to run on a local or Hadoop distributed environment [53]. The Pig-Latin language allows users to express their jobs as a series of operations and transformations to create the desired result. The

included operations include familiar database operations such as filtering, sorts and joins. A key advantage to Pig is that comparatively complex logic can be expressed in a small number of lines as users use a much higher level API. In a similar fashion to Hive, Pig automatically translates these operations into a series of MapReduce iterations [53].

- *Impala* – Apache Impala is a distributed massively parallel processing analytic database engine running on top of Hadoop [36]. The Impala database engine provides a mechanism for producing near real time queries on data stored within a Hive data store, leveraging HiveQL. An Impale service consists of three daemons; the core Impala daemon runs on each DataNode within a system, accepts distributes and coordinates user submitted queries across the cluster. The Impala Statestore monitors the core daemons and maintains state information, which is then redistributed to all core daemons. The Impala Catalog ensures metadata remains up to date across the entire cluster [36].
- *Storm* – Apache Storm is a distributed real-time computation system, based on the original Storm project create at Twitter [70]. Storm makes it easy to reliably process large amounts of streamed data, facilitating real time processing within the Hadoop ecosystem. Storm was designed so it can be used with any programming language through the use of Apache Thrift Topology Definitions. Although the base system is written in the Clojure functional programming language. Storm is comprised of two daemons, Nimbus and the Supervisor. Nimbus is the central orchestration process through which all jobs are delegated to Supervisors. Supervisors spawn and monitor worker threads to complete tasks. These daemons are fast-fail by design, meaning that worker processes are never affected by the failure of Nimbus or Supervisors. Storm has been designed to accept a data stream from a variety of sources including the Twitter streaming API, Apache Kafka, and JMS, although users can create their own stream receivers to accept data from almost any source. Similar to Dryad, Storm's computation model can be considered conceptually as a DAG, where data is streamed along the edges to computational vertices.
- *Giraph* – Apache Giraph is a system for large-scale graph processing, was developed as an open source project able to replicate the functionality of Google's Pregel [17,49]. Giraph processes graph-structured data in iterative *supersteps*. A *superstep* is an embarrassingly parallel problem where a single function is performed on each node within a graph. Changes made to nodes or edges within a *superstep* cascade through the graph on subsequent *supersteps*. Within any step a node can be marked as inactive, and processing continues until this is true for all nodes [32]. Giraph does not require any additional services and simply runs as MapReduce Jobs on standard Hadoop infrastructure.
- *Mahout* – Apache Mahout collection of machine learning libraries that are designed to be scalable and robust, with algorithms focusing in the areas of classification, clustering, filtering, data mining, evolutionary processes and dimensionality reduction [57]. All the original algorithms in Mahout were expressed as MapReduce applications; as such they require no extra daemon services. Recently Apache Mahout has expanded past MapReduce only implementations to include Spark and Flink as well.
- *Flink* – Apache Flink is a distributed data processing system designed with stream processing at its core [13]. Although the system is designed for stream based workloads it can also process batch workloads efficiently by treating them as a special form of a stream application. Users can create Apache Flink applications in either Java or Scala programming languages. Flink jobs can be run upon a Hadoop cluster using YARN, and can even execute Hadoop code from within a job.

14.4.2 HADOOP 2.0

In Hadoop V1, the management and resource negotiation is controlled via the MapReduce runtime. As such, any additional system wishing to access data stored in HDFS must be able to translate itself down to a series of MapReduce tasks. This places a massive restriction of the type of computation that can be performed via Hadoop. To address this, Hadoop 2.0 incorporated a new component called Yet Another Resource Negotiator (YARN) into the software stack [72]. YARN's key advancement is that it separates scheduling decisions from the data processing layer. This means that workflows no longer need to be constructed in terms of MapReduce tasks, potentially manipulated or misused, to achieve completion of work not suited to a MapReduce framework. Examples of this include map-only tasks written for Hadoop 1.0 that would spawn services such as web servers. One of the drawbacks with Hadoop V1 was the single point of failure that the JobTracker daemon represented. If a JobTracker were to fail, the resulting situation would be one where all running jobs within a system would be lost, there was no mechanism for automatic recovery from such a situation and all users would have to manually resubmit jobs for completion once the system was restored. YARN solves this key problem by removing cluster management duties from the JobTracker [72].

The YARN system architecture comprises three separate components, the ResourceManager (RM), the NodeManager (NM), and the ApplicationManager (AM) [72]:

- *ResourceManager* – The RM removes responsibility for scheduling of workflows from the JobTracker and serves containers for processing, which are logical bundles of available resources. This process is responsible for global system view collected from communication with the NM and AM, along with overall management of the system through the servicing resource requests made by user job submissions and the AMs. There is also a mechanism whereby the RM can request the return of resources from an AM, if it is deemed to be oversubscribed on the system, or forcibly claw back resources by instructing a NM to terminate containers if it deems too much time has elapsed following a request to an AM. In order to mitigate a single point of failure situation there is a standby RM that can take over responsibility of the system and spawn a new standby RM in the eventuality of a master RM failure. If there is a situation where a node, or NM, fails the RM will detect this, update system global state, report the failure to all running AMs and restart any AMs that were lost due to the failure of the NM.
- *NodeManager* – The NM launches and manages all containers, including AMs, configuring the environment as appropriate from AM, and user, requests received via the RM. Container requirements, including resources and environment, are described through a Container Launch Context (CLC) that are sent with every request for container creation. The NM is also responsible for reporting actual hardware utilization to the RM for maintenance of the global cluster state view. Upon creation all containers are issued a lease to use the requested resources this enables the RM to make better scheduling decisions based on a known lifetime of existing processes. Further, the NM cleans up all processes and files when a container exits, the NM is not aware is a container has exited cleanly or not. Handling of container failure is left to running application framework, i.e., it is up to the AM to know that a container has not exited cleanly and to request a new container via the RM.
- *ApplicationManager* – Bootstrap process, running within a NM spawned container, responsible for the execution of a "job" within the cluster, whether it be set of processes, a logical description of work, or even a long-running service. It requests required resources from the RM and executes code

within other available containers. Also responsible for local job optimization, management for loss of resources (node failure, other than node running AM) and reporting of job based metrics.

14.5 APACHE SPARK

Spark is a general-purpose parallel computing framework designed for computations of increased complexity to be performed upon massive datasets [78]. It's the first general-purpose compute platform to have emerged after the removal of cluster resource management from the MapReduce paradigm. The Spark computing framework grew from work at UC Berkeley and has quickly gained momentum within the data intensive computing community due to its performance and flexibility [9]. The speed increase is due, in part, to the Resilient Distributed Dataset (RDD) abstraction that allows working data be cached in memory, eliminating the need for costly intermediate stage disk writes [78]. At its core, Apache Spark is a cluster-computing platform that provides an API allowing users to create distributed applications, although it has grown to be the key component in the larger Berkley Data Analytics Stack (BDAS). Spark was designed to alleviate some of the constraints of the MapReduce programming model, specifically its poor performance when utilizing the same dataset for iterative compute processes due to its lack of an abstraction for distributed memory access [78]. Spark has been optimized for compute tasks that require the reuse of the same working dataset across multiple parallel operations, especially iterative machine learning and data analytic tasks, whilst also maintaining scalability and fault tolerance [64]. It has been argued that modern data analytic tasks that require the reuse of data are increasingly common in iterative machine learning and graph computation algorithms [77]. Examples of such algorithms include PageRank, used to create a rank of the popularity of web pages and other linked data sources and K-means clustering, used to group common members of a dataset together.

From the application developer's perspective, Spark allows for the creation of standalone programs in Java, Scala, R, and Python. Interestingly Spark also offers users the ability to utilize an interactive shell that runs atop of the cluster, behaving much like an interactive Python interpreter would. Spark has three key advantages for end users over MapReduce. Firstly, user's programs are less complicated to create and often require fewer lines of code. Secondly, users can create more complicated algorithms owing to the increased functionality of the API. Thirdly, due to Spark's speed advantage, user's computation will be completed in less time. Many of Spark's advantages are due to its more expressive and populated API. This higher-level API results in algorithm logic being expressed in fewer lines of code. The creators of Spark illustrate the brevity of Spark code by demonstrating that the word count example, which requires 50+ lines of Java MapReduce code, can be completed in just 3 lines of Scala code. (See Code 2.)

This is achieved via the use of Spark's higher level of API abstraction, allowing users access to a greater range of in-built functionality. Spark also contains concepts that will be familiar to many data scientists, such as data frames and SQL syntax. Compared with writing MapReduce code, Spark can be considered simpler due to the decreased number of nuances and caveats users have to consider during program creation. Another key advantage for end users is the runtime performance when using Spark, which has been shown to be greater than that of MapReduce [77]. In 2014, Spark beat the previous record held by Hadoop MapReduce for the GraySort competition. This competition requires frameworks to sort 100 TB of data in the shortest amount of time. Spark beat Hadoop's record by 49

```
01 val textFile = sc.textFile("hdfs://file.txt")
02 val counts = textFile.flatMap(line => line.split(" "))
              .map(word => (word, 1))
              .reduceByKey(_ + _)
03 counts.foreach(println)
```

CODE 2

Scala code for word count in spark

minutes, taking just a total of 23 minutes to sort the complete dataset [6]. Spark's victory is even more impressive when considering the compute resources consumed by both frameworks for their respective sorts. Spark utilized 1864 less compute nodes than Hadoop, requiring just 206 nodes to Hadoop's 2100 [6]. Other results have shown that Spark has achieved its stated aim of increasing the speed of iterative computing tasks. For example, Spark has been shown to be faster by up-to 20 times for K-means clustering [77] and up-to 8 times for PageRank [75].

From the administrator's perspective, Spark is also an attractive platform. The architecture of its design means that Spark is highly portable and can run on systems ranging from laptops, supercomputers, and the cloud. Spark also integrates well with existing Hadoop deployments, as it can be managed via YARN. In addition, Spark can be controlled via Mesos, an alternative cluster management framework developed by the same team as Spark [30]. Spark can also run in standalone mode, managing its own resources and scheduling. This mode can be configured to run on a single machine for testing and development work. For deployment to the cloud, each release of Spark contains scripts to manage a standalone deployment to Amazon's EC2 cloud. Although Spark's popularity has meant that all the major cloud vendors (Amazon's EC2, Microsoft's Azure and Google's Compute Engine) are offering prebuilt images containing Spark along with additional data intensive frameworks.

14.5.1 RESILIENT DISTRIBUTED DATASETS

The core of Spark is the Resilient Distributed Dataset (RDD) abstraction. An RDD is a read-only collection of data that can be partitioned across a subset of Spark cluster machines and form the main working component [77]. RDDs are so integral to the function of Spark that the entire Spark API can be considered to be a collection of operations to create, transform, and export RDDs. Every algorithm implemented in Spark is effectively a series of transformative operations performed upon data represented as an RDD. The key performance driver of Spark is that an RDD can be cached in memory of the Spark cluster compute nodes and thus can be reused by many iterative tasks. The input data that forms a RDD is partitioned into chunks and distributed across all the nodes in the Spark cluster, with each node then performing computation in parallel upon its own set of chucks. Physically an RDD is a Scala object and can be constructed from a variety of data sources, including files from HDFS or other file system, directly from Scala arrays or from a range of transformations that can be performed upon an existing RDD [77]. A key feature of RDDs is that they can be reconstructed if a RDD partition is lost using a concept called lineage and thus can be considered to be fault tolerant. Spark keeps a record of the lineage of an RDD but tracking the transformation that have been performed to create it. If any part of an RDD is lost then Spark will utilize this linage record to quickly and efficiently re-compute

the RDD using the identical operations that created the original version [77]. This method of lineage recomputation removes the need for costly data replication strategies used by other methods for abstracting in-memory storage across a compute cluster. However, if the lineage chain reaches a large enough size, users can manually flag a specific RDD to be check-pointed. Check-pointed RDDs are written to disk or HDFS to avoid the recomputation of long lineage chains. RDDs are not for storing or archiving final result data; this is still handled via HDFS or other file system.

There is a range of parallel operations that can be performed upon a RDD using the Spark Core API. These operations fall into two distinct categories, Transformations and Actions [77]. Transformations contain functions that will create new RDDs from existing data sources, whilst actions trigger computations to calculate a return value from the data or write the data out to external storage. The Transformation operations are lazy, so computation will not occur when these are called. An Action must be called to force the computation to happen. This lazy execution style has several advantages in the optimization of storage and performance for Spark. Spark will inspect a complete sequence of Transformations before a user's application is physically run upon the cluster. This allows for an overall picture of the data lineage to be created so that Spark can evaluate the complete transformation chain. Knowing this complete chain allows Spark to atomically optimize before runtime by only computing with the data required for the final result [9].

The RDD concept has further been expanded via the introduction of Spark DataFrames, [7]. A Spark DataFrame arranges the distributed collection of data into labeled columns similar to a traditional relational database. DataFrames are at a higher level of abstraction then RDDs and include a schema, allowing Spark to perform more automatic optimization at run time by exploiting the structured nature [7]. In addition to the performance improvements, the DataFrames API contains more domain specific functionality not offered by the RDD API, including joins and aggregations familiar to many data scientists.

14.5.2 DATA FLOW AND PROGRAMMING WITH SPARK

To create a program with Spark is very similar to existing data flow languages. End users are only required to write a single class that acts as a high-level program driver. This is in contrast to MapReduce, where 3 classes are required to create a typical program, the driver, mapper and reducer. In Spark, the user created driver program defines how the input data will be transformed to create the desired output. Users can create their applications in Java, Scala, Python or R using the provided APIs. However, Spark is a Scala first platform, meaning that new features in the API are added first in the Scala language. Scala is a functional language, and so can be a bit of a learning curve for users more familiar with OOP. Users of other languages are forced to wait for the latest Spark features to be pushed down steam. The program is submitted on the Spark cluster via the head node, and is then automatically distributed across the cluster so that it can be executed upon the partition of the RDD stored upon each worker node.

The data flow of a Spark program can be considered as follows: Firstly, the driver creates a single or multiple RDDs from a Hadoop or other data source. It then will apply transformations upon the RDD that will create new RDDs. Finally, some number of actions will be invoked upon the transformed RDDs to create the final output from the program. Perhaps the best way to envisage how Spark functions is to consider a typical example application. In this example we will explore how to create a simple log-processing program that will take a log file as input and count the number of times a key-

word appears. The overall plan for this application is to take a log file stored on a HDFS, extract only the required lines, load it into memory and count the number of stored lines. To create this application in Spark would require the following steps; firstly, an RDD would be created from the log file stored on the HDFS. The second RDD would be created via the filter transformation; this would filter the original file and only select the lines starting with the string "ERROR." A third RDD would be created using the map transformation to split the string on white space to remove the "ERROR" sub-string from the beginning and load it into memory. Finally, the number of elements would be counted using the inbuilt count function. It may seem inefficient to create three RDDs for such a simple application but as the computation is only triggered when the count Action is called, Spark's execution planning will ensure that only the required data is processed.

Users must still be careful when designing their algorithms to best make use of the Spark system. Spark can still require a fair amount of tuning to achieve optimal performance when running on a cluster.

14.5.3 SPARK PROCESSING ENGINES

In addition to the Spark Core API for creating data flow programs using the RDD abstract, Spark is integrated with a number of additional higher-level APIs that include functionality for specific classes of compute and data problems. As of Spark 2.0.1, the additional APIs are GraphX, MLlib, Spark SQL, and Spark Streaming. These APIs replicate functionality that, if using Hadoop, would require numerous additional domain specific packages to be deployed alongside the base Hadoop. As well as the standard libraries included with Spark, a large number of third-party libraries and APIs are maintained in an online repository and can easily be included by a user when creating their applications.

GraphX is a system that allows users to process graph data using Spark [75]. Its API includes some fundamental operations such as subgraph creation and a variant of Google's Pregel API, the first of the Think Like A Vertex (TLAV) systems [49]. The TLAV paradigm has emerged as the most widely used method for processing massive graphs on distributed systems [51]. GraphX also includes implementations of many of the most commonly used algorithms for graph analysis including PageRank, connected components and triangle counting. As many graph-based algorithms are inherently iterative processes, often values have to be recomputed on the vertices until a particular threshold is reached, they are an ideal class of problems to be performed via Spark. GraphX's performance has been shown to beat other TLAV style frameworks based on Hadoop, as well as other dedicated graph processing systems [8]. A detailed comparison of some of the major graph processing platforms, performed by Batarfi et al. [8], shows that GraphX is often the faster platform at performing the computation, while also consuming fewer resources and generating less network traffic to do so.

MLlib is Spark's distributed machine learning library and provides fast implementations of several key algorithms from the field [52]. Using the DataFrame API, users can create a machine-learning pipeline, mixing several algorithms together to produce the required result. MLlib includes many of the fundamental algorithms of machine learning including linear support vector machines (SVM), logistical regression, random forest and several clustering methods along others [52]. MLlib is shown to be faster and more scalable then its main, Hadoop based, rival Mahout [52].

The Spark SQL library allows for Spark to process and query structured data via standard SQL syntax [7]. Spark SQL can load and query data taken from a variety of sources, and due to sharing some components, is particularly well integrated with Hive. Data can be loaded from mutual sources

into a DataFrame, upon which a query can be run. Spark SQL supports all the main SQL data types including bools, integers, floats and strings. Perhaps the most interesting aspect to Spark SQL is that it allows users to combine relational with procedural processing within the same application [7]. Users can, for example, feed the results from an SQL query into functions from the RDD API to create complicated data flows. Spark SQL simultaneously makes Spark more accessible to new users and introduces key optimizations, in the form of DataFrames, for existing ones [7].

The last of the integrated Spark processing engines is Spark Streaming, which allows users to create applications that process data in near real-time [79]. The API has been designed such that writing a stream application is very similar to creating a batch processing application. Users can reuse code from batch processing and even integrate data from historical sources, via the use of join, for example, as the streaming application is running. An application created using Spark Streaming can take input from a variety of sources in including HDFS, Twitter Flume and ZeroMQ. Spark Streaming utilizes the DStreams abstract to represent input stream data as a series of RDDs [79]. Due to the process of creating the RDD's Spark streaming is not real time but can be considered as a micro-batch system.

The DStreams abstraction partitions the input stream into a series of time blocks. Each block contains a few 100 milliseconds of the stream and is stored in an RDD, upon which the standard Spark batch functions can be performed. The performance of Spark Streaming has been shown to scale nearly linearly, and can achieve a throughput rate of 20 MB per second on each node of the Spark cluster [79].

These additional integrated libraries included with Spark are another of its key advantages and set it apart from the Hadoop Ecosystem. Firstly, users need only learn one programming language and set of system APIs. This also benefits system administrators who only have to deploy and configure one software package, versus the many specialized systems that it would require to replicate the same functionality. Secondly, the integration and interoperability enabled between different computing paradigms via the unified Spark stack, allows users to create new classes of applications. This new class of big data applications will be made possible via the integration of previously separate systems. One such example application enabled by this integration is real-time fraud detection for the financial sector. Creating this application in Spark, the user could train the MLlib module on historical data and then use the developed model to detect fraud happening in real-time using the Spark Streaming module. This is just one of the many new applications that could be created using the integrated platform offered via the Spark framework.

14.5.4 HADOOP ECOSYSTEM TAXONOMY

Table 14.2 gives a summary of some of the key differences between, Hadoop V1, Hadoop 2.0, Spark, and Flink. These frameworks are compared on a number of metrics and capabilities.

14.6 THE ROLE OF CLOUD COMPUTING

When comparing the computational requirements for big data systems against traditional high performance computers (HPC), the economic and technological barrier to entry was obviously lower. Both paradigms can make use of commodity systems for proof-of-concept and sometimes as basic implementations for production systems [31,22]. However, to truly scale HPC systems one needs specialist networking with low latency and high bandwidth. As HPCs run on the principle of a single system

Table 14.2 Comparison of various big data ecosystems

	Hadoop	Hadoop 2	Spark	Flink
Execution Model	Batch	Batch, Interactive	Batch, Interactive, Stream	Stream, Batch
Language	Java	Java	Scala, Java, Python, R	Java, Scala
Data Locality	Disk	Disk	Memory, Disk	Memory, Disk
Data Partition	HDFS	HDFS	RDD	HDFS
Data Storage	HDFS	HDFS	HDFS, S3, Tachyon	HDFS, Tachyon, Hbase
Machine Learning	Mahout	Mahout	MLlib	FlinkML
Graph Analytics	Giraph	Giraph	GraphX	Gelly
Streaming	None	Storm	Native	Native
Database	Hive, Hbase	Hive, Hbase	Spark SQL	Table API
Fault Tolerance	Disk Write	Disk Write	Lineage	Checkpoints
Programming Model	Directed Graph	Directed Graph	Directed Acyclic Graph	Directed Acyclic Graph
Security	Access Control List	Kerberos	Kerberos (via YARN)	Access Control List
Bottlenecks	Disk IO, Network IO	Disk IO, Network IO	Memory BW, Network IO	Disk IO, Network IO

image, the performance of a distant processor is expected to be the same or as close to the same as the processor interacting with a user. When a task is initiated, all participatory processing elements (PEs) are required to operate in a timely manner and the occurrence of inter-element dependencies are the norm.

While big data systems are still constrained by Amdahl's Law,[1] systems running Hadoop or Spark type frameworks do not face the type of bottlenecks outlined above. Due to the almost bag-of-tasks approach and greedy scheduling where the same task may be replicated to maximize throughput, performance is very much dependent on the configuration of the PE. At the processing element level, faster CPUs are beneficial but not mandatory. Like any other computing paradigm, the more memory and the faster it is, the better; however, the best commodity memory can deliver great results. As discussed earlier, where big data processing elements differ from their HPC counterparts is the inclusion of high volume and high speed directly attached storage. The penetration of solid-state storage devices has meant that on a performance basis *server grade* hardware and commodity hardware are almost equally matched. With easy and affordable availability of 10 gigabit Ethernet based networking within the commodity market, networking is no longer a bottleneck for organizations implementing local big data systems.

For an organization aiming to incorporate data-centric approaches into their business and product processes, certain other factors play a role in affordability. For a 24×7 operational system, with multiple users and operators that is continuously ingesting, processing and delivering data, server grade equipment becomes a necessity. This sort of hardware is designed with resilience and robustness in

[1] "The maximum speed up (the maximum number of processors which can be used effectively) is the inverse of the fraction of time the task must proceed on a single thread" [61].

mind for round the clock operation. Server grade hardware, in terms of performance, is matched or slightly better (ECC Memory) than commodity items, but they guarantee reliability. There is a higher price tag attached to this reliability, but it does not just come in the form of capital expenditure on the equipment itself. Server grade hardware requires specialist infrastructure in the form of uninterrupted power supplies with power shaping, industrial cooling units, and server rooms and racks to host the machines. Along with the capital there is a higher operational expense as well. Qualified staff and expensive maintenance contracts are required to manage the infrastructure. As the company expands its data-driven environment, the electrical costs also skyrocket.

To achieve scale, resilience, and robustness, cloud computing provides a cost optimal solution. Within cloud environments (like Amazon's Elastic Compute Cloud) optimized configurations can be found for all the popular data analytics platforms. The developers of the data analytics platforms have configured many of the available images, ensuring the highest quality of the available implementation. Scaling is simpler as the costs do not include capital expenditures for infrastructure such as UPS, rack space, or networking. Operational costs such as electricity and cooling demands also do not change for end user. Table 14.3 shows the different offerings from popular cloud providers.

The pay-per-use model allows for organizations to scale up or down, as the business requires, thus ensuring no waste of money. However, there are some cost considerations that need to be taken into account. For many cloud providers the adage "compute is cheap but bandwidth is expensive" holds true. The phrase refers to the cost in time to move data between peripherals, storage, and the CPU, but it applies to creating compute systems in the cloud and moving data from your local system to the cloud. If an organization needs a large scale big data system for 10 days a month, then for the remaining days it needs to consider the cost of leaving the system running or reprovisioning the whole system when needed and retransferring the data. Cloud services that charge extra for either internode network data traffic make jobs in Hadoop more financially expensive than in Spark, while those that charge more for data at *ingress* or *egress* points may not be as optimal for continuous tear down approaches. Some cloud providers charge extra for HDFS optimized instances, but then exclude internode communications.

The main consideration an organization needs to make, and one of the biggest stumbling blocks, before cloud based deployment of big data solutions is data security and privacy. Depending on the nature of the data, its associated data protection levels, and any service or contractual agreements in place, the public nature of the cloud can be a nonstarter. While the cloud was imagined to be this amorphous collection of resources, in practice they exist across different geo-political boundaries and are thus subjected to a myriad of legal provisions. The United States Patriot Act is a prime example of legislation affecting data privacy [55]. When dealing with customer data organizations can write in clauses in contracts enabling cloud storage of collected information but historic information is still restricted and potentially cannot be included into any data processing framework.

Not every company that needs to develop a model of customer interaction can afford to host its own big bata system. The earlier example of a smart home that aggregates copious amounts of sensor data and then simulates the different possibilities of how the day will proceed would need both a big data system and an HPC cluster. The first is to process the data and create the models, while the second to execute the simulations. Clearly, the homes of the future will not have a data intensive or HPC cluster (or both) in the basement. Cloud computing will be the glue that links the back-office analytics to the real world.

Table 14.3 Offerings of big data on public clouds [29]

	Google	Microsoft	Amazon	Cloudera
Big data storage	Google cloud services	Azure	S3	
MapReduce	AppEngine	Hadoop on Azure	Elastic MapReduce (Hadoop)	MapReduce YARN
Big data analytics	BigQuery	Hadoop on Azure	Elastic MapReduce (Hadoop)	Elastic MapReduce (Hadoop)
Relational database	Cloud SQL	SQL Azure	MySQL or Oracle	MySQL, Oracle, PostgreSQL
NoSQL database	AppEngine Datastore	Table storage	DynamoDB	Apache Accumulo
Streaming processing	Search API	Streaminsight	Nothing prepackaged	Apache Spark
Machine learning	Prediction API	Hadoop + Mahout	Hadoop + Mahout	Hadoop + Oryx
Data import	Network	Network	Network	Network
Data sources	A few sample datasets	Windows Azure marketplace	Public Datasets	Public Datasets
Availability	Some services in private beta	Some services in private beta	Public production	Industries

14.7 THE FUTURE OF BIG DATA PLATFORMS

14.7.1 BIG DATA APPLICATIONS

The use of big data techniques and processing has already been successful in revolutionizing several fields. Areas such as social network analysis, the advertising industry and the management of large computer systems have already benefited. However, big data is poised to be successful in many more including:

- *Healthcare* – There are numerous ways that big data could revolutionize the Healthcare industry. One such area that has been explored is error checking and anomaly detection in medical research datasets [10], although the possibilities for advances across the whole field are endless [59].
- *Governance* – Extending the concept of creating holistic user models, governments are attempting to adopt a person-centric approach to governance. This will lead to customized interactions between the citizenry and local authorities [40]. Through analytics, modeling and simulations, urban authorities hope to be proactive in dealing with potential difficulties rather than reactive [48].
- *Smart Cities* – With the automation of many core operations across urban environments and the deployment of different kinds of sensors where possible, the next step is the integration of these systems. Complex models are being developed that integrate real-time data from the various systems and sensors with simulated environments and humans-in-the-loop to streamline city wide operations and mitigate knock on affects that are caused by problems within one or more systems [19,62].
- *Industry 4.0* – The next revolution in the industrial production processes is the development of agile and flexible systems that are driven by analytics and simulation to respond quickly to problems and variations within the production environment. Intelligent Manufacturing Systems (IMS), using virtual, fractal, bionic, and holonic manufacturing systems, are being developed to provide production

control. The systems not only factor in the factory floor but also the wider supply chain within its decision models [25,41].
- *The Internet of Things* (*IoT*) – The IoT revolution will be driven by integration of Internet functionality into a range of products and sensors, allowing them to record and transmit data to be sent for analysis [26]. It can be argued that IoT will be the technological backbone used to drive the new class of intelligent Smart Cities and Industry 4.0 applications [26].

To meet the demands of these fields, future big data applications will need ever-greater integration with machine learning to drive intelligent conclusions from the wealth of available data. Many big data applications have traditionally been batch processes, where data was first collected and stored, before being processed at a later date. However, IoT, Healthcare, and Smart Cities applications will need to process data in real-time as they will require instantaneous answers drawn from the stream of data. In addition, all areas will begin to demand subsecond responses to queries on existing data sources.

14.7.2 BIG DATA FRAMEWORKS AND HARDWARE

The evolution of data intensive frameworks has been driven by the roles demanded of them by data intensive applications. The development of the original MapReduce was in response to a new type of workload. Spark was created, in part, as a way to perform many iterative computations over the same working dataset. As such, the role of data intensive frameworks has been evolving since the introduction of MapReduce in 2004, driven by new classes of data intensive problems. Some of the current key roles of a data intensive framework include:

- Provide fault tolerant storage for massive quantities of data from disperse sources.
- Provide a common and expressive set of APIs, written in a variety of languages, enabling users to access and process massive quantities of data.
- Allow users to focus on application development by abstracting away the traditional complexities of parallel computing.
- Automatically schedule jobs to run in parallel across the underlying compute resource, whilst considering aspects such as data locality.

The evolution of data intensive frameworks will continue and will be focused towards increasing the speed of applications, creating more scalable applications and creating easier to use APIs with greater breadth of functionalities. More specifically, data intensive framework evolution will focus on these key areas:

- Continue to increase the possible application performance to meet the demand of sub-second and real-time data analytics by more optimal use of resources and by allowing the applications to scale across larger compute clusters.
- Allow an increasingly diverse range of compute paradigms to interoperate and share access to data and allow the same physical resource to be shared between compute and data centric frameworks.
- Continue to reduce the complexity of creating big data applications with the introduction of more generic and higher-level APIs for developers, along with creating easy deployment methods for data intensive frameworks.

14.7.2.1 Performance and scalability increases

The original performance driver of MapReduce was disk-based data locality and enabling its central philosophy – bring the compute to the data. While this approach was clearly very successful, the need for ever-greater performance, driven by the need for real time analytics, has meant that researchers are looking for new ways to speed-up big data frameworks. There is an argument that network technology has improved to such a degree that it is now able to outpace the speed of local storage. As such, some have claimed that disk-based locality can now be considered irrelevant, as reading from a remote disk is now as fast as local storage due to the modern high-speed networks found in datacenters [2]. While the introduction of technologies like PCIe based NVMe Solid State Disk (SSD) storage will mean that this is not universally true, it can be argued that most data intensive clusters are still using traditional spinning disks due to their lower cost and large capacity [2]. The need for greater performance forced the developers of Spark to abandon disk locality in favor of memory locality. The move to memory-based locality has led to up to an order of magnitude increase in performance over the disk-based Hadoop system [77]. Now that data intensive frameworks are exploiting in-memory processing, future increases in performance will have to come from elsewhere as memory is the currently the fastest storage. It is highly unlikely that CPU caches will increase to a size that can accommodate modern massive datasets. So where is it likely that future increases in performance and scalability will come from? Three possible areas are the use of in-memory storage systems, specialized hardware and continued software improvements.

Whilst there is a move to using memory for dataset processing, current distributed file systems, such as HDFS, still rely on hard drive for the data storage and archiving. Systems such as Spark still have to read input data from disk into main memory before processing can begin, leaving hard drive transfer rates as a bottleneck in application performance. However, there have been efforts to move the storage system for data intensive frameworks into system memory. One example of such system is Alluxio, which aims to offer a fault-tolerant distributed file system, using the speed offered by running elements of itself in-memory to massively increase file write and read times [45]. While Alluxio is not a complete in-memory system, it still utilizes a disk based file system such as HDFS or GlusterFS to provide a persistence copy of the data, it provides a massive performance increase over using those systems alone. Alluxio uses a linage concept, similar to that of Spark's RDD model, to avoid the need for costly data replication strategies, instead the linage of tasks that created a certain dataset is tracked so it can be recomputed on demand [45]. Going forward, there will continue to be developments towards distributed in-memory file systems that are both data locality aware and fault tolerant.

As data intensive frameworks are turning to memory as method for increasing performance, the underlying hardware will need to adapt. One aspect of computer hardware, which has yet to see much adoption for big data workloads, is the Graphics Processing Unit (GPU) [63]. GPUs were first introduced as coprocessors to compute the demanding workload of graphics for video games. However, it was soon noted that they possess characteristics that make them ideal for certain compute intensive workloads. This notion of using a GPU for tasks other than graphics processing is known as General-Purpose computation on Graphics Processing Units (GPGPU). Technologies such as CUDA and OpenCL now allow users direct access to perform their computations upon modern powerful graphics hardware. However, for big data workloads that utilize massive quantities of data, GPUs have some limitations that make them less than ideal [63]. GPUs are unable to directly access data stored on disk or in memory on the host machine, so all required data must be transferred to the memory of the GPU

card via the PCIe bus. The PCIe bus has lower bandwidth than memory, so transferring large quantities of data over it can be considered a bottleneck. To compound this problem, GPUs have much smaller amount of memory than is commonly found in big data compute nodes, often over an order of magnitude less, meaning data has to be shuffled in sections. Due to these problems, GPUs have seen little adoption in big data workloads. However, as data intensive workloads diversify and start to include an increasing amount of simulation and machine learning aspects, both of which are classes of computation that GPUs excel at performing, GPUs could become a powerful way to increase performance. GPUs are just one of the forms of coprocessor that could be used to accelerate certain aspects of the modern big data workload, others include FPGAs and the Intel Xeon Phi.

The software used by data intensive frameworks will continue to improve, making better use of the underlying hardware. It has been well established that the bottleneck for many in-memory applications is data transfer over the network [77]. New advances in technology will have to find ways to improve this to further increase performance. Another aspect that future systems could explore is exploiting a greater knowledge of the workload requirements. Having a deeper understanding could enable the system to make intelligent decisions about data preprocessing and hardware scheduling to better optimize resources.

14.7.2.2 Diversification of compute paradigms

MapReduce was originally introduced to perform a specific class of computational problem. Since its introduction the model has been adapted to many diverse computing tasks such as machine learning and graph processing. However, early attempts at porting other paradigms to MapReduce required that they be expressed as a series of MapReduce tasks, limiting functionality. More recently there has been a move away from MapReduce type computation, on data stored in HDFS like file systems. The introduction of YARN addressed this issue by removing the resource negotiation and scheduling from the compute engine, thus MapReduce is now just one of many frameworks that can access data stored in the DFS. As such, systems like Tez and HBase no longer have to abstract down to a series of MapReduce tasks to complete a workflow.

Whilst MapReduce has been very successful at dealing with fault tolerance, data locality, and the obscuring of parallel computing complexities, it is clear that it is not the best approach for every workload. Other frameworks now need to take the successes of the MapReduce framework and apply it to other, more general, data intensive workloads. The move to processing in-memory has increased the number of possible workloads massively, with Spark being able to offer many different compute paradigms. The unification of different compute functionality offered by a system like Spark allows new classes of application to be developed. Such applications could create advanced workflows by, for example, allowing data from relational queries to be fed into graph processing algorithms, with the final results analyzed by machine learning, all within the same framework. To further exploit this potential and enable the creation of the next generation of data intensive applications, future frameworks will need to increase the capability of current implementations and extend the number of available compute paradigms. It has been argued that modern data analytics needs to include simulation and what-if analysis to gain a deeper insight into the data [28]. Future data intensive frameworks will need to include and interoperate with more complex compute focused methods including deep learning and data driven simulation.

14.7.2.3 Simplifying data centric development

Big data processing is typically done on large clusters of shared-nothing commodity machines. One of the key lessons from MapReduce is that it is imperative to develop a programming model that hides the complexity of the underlying system, but provides flexibility by allowing users to extend functionality to meet a variety of computational requirements. Whilst a MapReduce application, when compared with an MPI application, is less complex to create, it can still require a significant amount of coding effort. As data intestine frameworks have evolved, there have been increasing amounts of higher-level APIs which are designed to further decrease the complexities of creating data intensive applications. Current data intensive frameworks, such as Spark, have been very successful at reducing the required amount of code to create a specific application. Future data intensive framework APIs will continue to improve in four key areas; exposing more optimal routines to users, allowing transparent access to disparate data sources, the use of graphical user interfaces (GUI) and allowing interoperability between heterogeneous hardware resources.

- Future higher-level APIs will continue to allow data intensive frameworks to expose optimized routines to application developers, enabling increased performance with minimal effort from the end user. Systems like Spark's Dataframe API have proved that, with careful design, a high-level API can decrease complexity for user while massively increasing performance over lower-level APIs.
- Future big data application will require access to an increasingly diverse range data sources. Future APIs will need to hide this complexity from the end user and allow seamless integration of different data sources (structured and semi- or nonstructured data) being read from a range of locations (HDFS, Stream sources and Databases).
- One, relatively unexplored, way to lower the barrier of entry to data intensive computing is the creation of GUIs to allow users without programming or query writing experience access to data intensive frameworks. The use of a GUI also raises other interesting possibilities such as real time interaction and visualization of datasets.
- APIs will also need to continue to develop in order to hide the complexities of increasingly heterogeneous hardware. If coprocessors are to be used in future big data machines, the data intensive framework APIs will, ideally, hide this from the end user. Users should be able to write their application code, and the framework would select the most appropriate hardware to run it upon. This could also include pushing all or part of the workload into the cloud as needed.

For system administrators, the deployment of data intensive frameworks onto computer hardware can still be a complicated process, especially if an extensive stack is required. Future research is required to investigate methods to atomically deploy a modern big data stack onto computer hardware. These systems should also set and optimize the myriad of configuration parameters that can have a large impact on system performance. One early attempt in this direction is Apache Ambari, although further works still needs under taking, such as integration of the system with cloud infrastructure. Could a system of this type automatically deploy a custom data intensive software stack onto the cloud when a local resource became full and run applications in tandem with the local resource?

14.7.2.4 Physical resource sharing

Computer clusters represent a large investment for any institution that decides they want to perform big data analytics in house rather than in the cloud. Unfortunately, the 3 major paradigms, (HPC, Big Data and Distributed RDBMS) do not share the same physical resource well. The key issue here is that current systems do not share a single scheduler, so are unaware of when a competing framework might be using resources. This is obviously problematic if more than one framework is needed. Institutions have three possible ways of dealing with this issue; run physically separate dedicated resources, separate part of the same physical resource up into dedicated sections or, lastly, by running the various software stacks in a virtualized environment. These methods all have issues, meaning that none are an ideal solution. Purchasing and maintaining separate resources would massively increase expense. Partitioning a single resource could lead utilization and load balancing problems, particularly if one framework is more commonly used. Lastly, virtualizing the software stack could result in less optimal use of the hardware due the overheads inherent in the virtualization process.

To solve this problem, there is currently a great deal of research to develop systems that will allow multiple competing compute frameworks to share the same underlying resource. One such system is Apache Mesos, which allows both Hadoop and MPI to run alongside one another [30]. Mesos works by sitting between the respective frameworks own schedulers and the hardware, deciding which compute resources to offer up to each framework. When a user submits a MapReduce application, for example, the Hadoop scheduler will inform the Mesos master process that it needs some resources. The Mesos master will then decide what of the available resources to offer to Hadoop. If the offer of resources from Mesos does not meet the requirements of the Hadoop job, it can reject the offer and wait for a more suitable offer to be made. If the offer is accepted, Mesos will launch executor tasks to perform the computation. The executor tasks run inside of containers to isolate them from one another [30]. In this way, Mesos acts as the overall controller of compute cluster but still allows the individual frameworks to operate in their own optimal and unique way. Unfortunately, Mesos does not run with standard implementations of the frameworks, as they need additional code to interoperate with the Mesos scheduler. Although the most popular frameworks including Hadoop, Spark and MPICH2 already have Mesos compatible builds available. Mesos is just one such system to tackle the problem; others include Google's Omega [65] and Borg [73] and Microsoft's Apollo [11]. Although the open source nature of Mesos has meant that it has seen good adoption within industry [5]. Whilst Mesos is a good starting position, future work will be needed to expand the system. Such work could include the integration of cloud resources for running containers, more sophisticated resource offers, the integration of more frameworks and support for a more diverse range of hardware.

14.7.3 BIG DATA ON THE ROAD TO EXASCALE

The increase in available data inputs led to the creation of data centric processing frameworks. Traditional large computer systems were geared to scale the processing elements but not increasing input data. This trend does not appear to be changing. In its 2015 Hype Cycle for Emerging Technology, Gartner Inc. put IoT, Advanced Analytics and Machine Learning at the "Peak of Inflated Expectations" with 5–10 years still to go before plateauing out as productive technologies [23]. Over the next 10 years and beyond the size and scope of data will grow considerably and computing paradigms will also have to adapt.

On the technological front, exascale has captured the imagination of experts in the field. Exascale has two facets, the first and the one that gets the most hype is exascale computer systems. Computers capable of calculating 10^{18} floating-point operations a second, or an exaflop, is the next challenge for computing in general and high performance computing in particular. This scale of computing comes with a whole host of challenges: from limitations of the silicon substrate, to the power and thermal limits, and I/O limitations (disk, network, etc.) breaking the exa barrier will be challenging. On the software side as well, compilers and program workflows have not fully addressed the challenges of petascale (10^{15}) computing, and so exascale remains a goal that keeps getting pushed back.

On the data front, reaching exascale is right on the horizon. Datasets that are petabytes and terabytes in size are currently being processed by teraflop and gigaflop capable systems. With the step changes coming, through the phenomenon of Internet of things and person-centric services, workloads that are 1 exabyte (EB) in size will quickly become a reality. Exabyte's of data will be thrust upon businesses and this will be a real challenge to deal with. At that scale the noise ratio becomes much higher and the cost trade-offs also need to be revaluated.

The challenges facing the high performance computing community are the same that will shape the direction of big data. Technologically, scaling big data frameworks to exaflop and petaflop systems will see the same compiler, networking, power, and cost issues as in HPC. Considerable development in both hardware and software is required to deal with the oncoming data deluge.

14.8 CONCLUSION

This chapter has explored the evolution of big data technologies. Since the introduction of MapReduce, the data intensive computing world has been evolving rapidly. The original, inflexible, MapReduce programming model has been expanded to incorporate the full spectrum of data intensive computing paradigms. Now that access to the data stored in a distributed file system has been compartmentalized and separated from the MapReduce programming model, new systems like Apache Spark have been able to push application performance further by utilizing memory locality. MapReduce was only the first step in the democratization of parallel computing, future frameworks will continue this trend and make massive compute power available to nonspecialists. With in-memory processing fully integrated in the big data stack, improving framework performance will require more intelligence from the compilers and middleware to make better use of the available hardware. The move to in-memory computation has also expanded the number of paradigms these data intensive frameworks are able to perform, as the ability to reuse the same working dataset is ideal for both Machine learning and Graph processing. The ability to utilize multiple computing paradigms within the same application, will result in a new generation of data intensive applications being created.

In terms of the possibilities for these applications, returning to the earlier smart home example, in this globally connected world Owner A may be rewarded with an extra strong coffee at his/her 4 p.m. meeting courtesy of a *discussion* between the coffee maker at the meeting place and the home owners home agent that takes place in the cloud and is triggered by the facial recognition system on a CCTV camera.

REFERENCES

[1] F.N. Afrati, J.D. Ullman, Optimizing multiway joins in a map-reduce environment, IEEE Trans. Knowl. Data Eng. 23 (9) (2011) 1282–1298, http://dx.doi.org/10.1109/TKDE.2011.47.

[2] G. Ananthanarayanan, A. Ghodsi, S. Shenker, I. Stoica, Disk-locality in datacenter computing considered irrelevant, in: HotOS'13 Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems, 2011, pp. 1–5. Retrieved from http://dl.acm.org/citation.cfm?id=1991596.1991613.

[3] M. Ali-ud-din Khan, M.F. Uddin, N. Gupta, Seven V's of big data understanding big data to extract value, in: 2014 Zone 1 Conference of the American Society for Engineering Education (ASEE Zone 1), IEEE, April 2014, pp. 1–5.

[4] Apache, Hadoop, http://hadoop.apache.org, 2009.

[5] Apache, https://mesos.apache.org/documentation/latest/powered-by-mesos/, 2016.

[6] M. Armbrust, T. Das, A. Davidson, A. Ghodsi, A. Or, J. Rosen, R. Xin, et al., Scaling spark in the real world: performance and usability, Proc. VLDB Endow. 8 (12) (2015) 1840–1843, http://dx.doi.org/10.14778/2824032.2824080.

[7] M. Armbrust, R. Xin, M. Zaharia, Spark SQL: relational data processing in spark, in: SIGMOD'15, 2015.

[8] O. Batarfi, R. El Shawi, A.G. Fayoumi, R. Nouri, S.-M.-R. Beheshti, A. Barnawi, S. Sakr, Large scale graph processing systems: survey and an experimental evaluation, Clust. Comput. (2015), http://dx.doi.org/10.1007/s10586-015-0472-6.

[9] B. Bengfort, J. Kim, Data Analytics With Hadoop, Oreilly & Associates Inc, 2015.

[10] S. Bonner, A.S. McGough, I. Kureshi, J. Brennan, G. Theodoropoulos, L. Moss, G. Antoniou, et al., Data quality assessment and anomaly detection via map/reduce and linked data: a case study in the medical domain, in: 2015 IEEE International Conference on Big Data (Big Data), IEEE, 2015, pp. 737–746.

[11] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, L. Zhou, et al., Apollo: scalable and coordinated scheduling for cloud-scale computing, in: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), 2014, pp. 285–300.

[12] M. Cafarella, D. Cutting, Apache HBase. Retrieved from https://hbase.apache.org/, 2007.

[13] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas, Apache flink: stream and batch processing in a single engine, Data Eng. 28 (2015).

[14] Cassandra, from http://cassandra.apache.org/, 2010.

[15] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, R.E. Gruber, et al., BigTable: a distributed storage system for structured data, in: 7th Symposium on Operating Systems Design and Implementation (OSDI'06), November 6–8, Seattle, WA, USA, 2006, pp. 205–218.

[16] P. Chen, C.-Y. Zhang, Data-intensive applications, challenges, techniques and technologies: a survey on big data, Inf. Sci. 275 (2014) 314–347, http://dx.doi.org/10.1016/j.ins.2014.01.015.

[17] A. Ching, C. Kunz, Apache Giraph. Retrieved from https://giraph.apache.org, 2013.

[18] M. Cox, D. Ellsworth, Application-controlled demand paging for out-of-core visualization, in: Proceedings of the 8th Conference on Visualization'97, IEEE Computer Society Press, 1997 (pp. 235-ff).

[19] M. Deakin, The embedded intelligence of smart cities, Intell. Build. Int. 3 (3) (2011) 189–197.

[20] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Commun. ACM 51 (2004) 1–13, http://dx.doi.org/10.1145/1327452.1327492.

[21] D. DeWitt, M. Stonebraker, MapReduce: a major step backwards, Database Column 44 (2008) 1–3. Retrieved from http://codepaint.kaist.ac.kr/wp-content/uploads/2014/03/MapReduce.pdf.

[22] D. Eadline, Hadoop 2 Quick-Start Guide: Learn the Essentials of Big Data Computing in the Apache Hadoop 2 Ecosystem, Addison–Wesley Professional, 2015.

[23] Gartner, Newsroom Gartner's 2015 hype cycle for emerging technologies identifies the computing innovations that organizations should monitor. Retrieved from http://www.gartner.com/newsroom/id/31, 2015.

[24] S. Ghemawat, H. Gobioff, S.T. Leung, The Google file system, Oper. Syst. Rev. 37 (5) (2003) 29–43.

[25] W.A. Gruver, D.B. Kotak, E.H. van Leeuwen, D. Norrie, Holonic manufacturing systems: phase II, in: Holonic and Multi-Agent Systems for Manufacturing, Springer, Berlin, Heidelberg, 2003, pp. 1–14.

[26] J. Gubbi, R. Buyya, S. Marusic, M. Palaniswami, Internet of Things (IoT): a vision, architectural elements, and future directions, Future Gener. Comput. Syst. 29 (7) (2013) 1645–1660, http://dx.doi.org/10.1016/j.future.2013.01.010.

[27] Z. Guo, G. Fox, M. Zhou, Investigation of data locality in MapReduce, in: Proceedings – 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2012, 2012, pp. 419–426.

[28] P.J. Haas, P.P. Maglio, P.G. Selinger, W. Tan, Data is dead ... without what-if models, Proc. VLDB Endow. 4 (12) (2011) 1486–1489, http://doi.org/10.1002/num.20576.

[29] I.A.T. Hashem, I. Yaqoob, N.B. Anuar, S. Mokhtar, A. Gani, S.U. Khan, The rise of "big data" on cloud computing: review and open research issues, Inf. Sci. 47 (2015) 98–115.
[30] B. Hindman, A. Konwinski, A. Platform, F.-G. Resource, M. Zaharia, Mesos: a platform for fine-grained resource sharing in the data center, Proc. NSDI 32 (2011). Retrieved from http://static.usenix.org/events/nsdi11/tech/full_papers/Hindman_new.pdf.
[31] V. Holmes, I. Kureshi, Huddersfield university campus grid: QGG of OSCAR clusters, J. Phys. Conf. Ser. 256 (1) (2010).
[32] S. Sherif, Processing large-scale graph data: a guide to current technology, IBM Developerworks, 2013, pp. 1–13.
[33] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad: distributed data-parallel programs from sequential building blocks, Oper. Syst. Rev. (2007) 59–72, http://dx.doi.org/10.1145/1272996.1273005.
[34] S. Jha, J. Qiu, A. Luckow, P. Mantha, G.C. Fox, A tale of two data-intensive paradigms: applications, abstractions, and architectures, Big Data 2 (1) (2014) 8. Distributed, Parallel, and Cluster Computing. Retrieved from http://arxiv.org/abs/1403.1528.
[35] S.D.T. Kelly, N.K. Suryadevara, S.C. Mukhopadhyay, Towards the implementation of IoT for environmental condition monitoring in homes, IEEE Sens. J. 13 (10) (2013) 3846–3853.
[36] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, M. Yoder, et al., Impala: a modern, open-source SQL engine for Hadoop, in: 7th Biennial Conference on Innovative Data Systems Research (CIDR'15), 2015.
[37] R.T. Kouzes, G.A. Anderson, S.T. Elbert, I. Gorton, D.K. Gracio, The changing paradigm of data-intensive computing, Computer 42 (2009) 26–34, http://dx.doi.org/10.1109/MC.2009.26.
[38] A. Lakshman, P. Malik, Cassandra, Oper. Syst. Rev. 44 (2) (2010) 35, http://dx.doi.org/10.1145/1773912.1773922.
[39] D. Laney, 3D data management: controlling data volume, velocity and variety, META Group Res. Note 6 (2001) 70.
[40] L. Larroquette, S. Srivastava, Citizen centric governance, J. Dev. Manag. 1 (4) (2013) 439.
[41] J. Lee, B. Bagheri, H.A. Kao, A cyber-physical systems architecture for industry 4.0-based manufacturing systems, Manuf. Lett. 3 (2015) 18–23.
[42] J. Leskovec, A. Rajaraman, J.D. Ullman, Mining of Massive Datasets, Cambridge University Press, 2014.
[43] LHC, CERN Accelerating Science. Processing: What to Record? N.p., 2015. Web. 24 Feb. 2016.
[44] F. Li, B. Ooi, M. Ozsu, S. Wu, Distributed data management using MapReduce, ACM Comput. Surv. 46 (3) (2013) 31:1–31:42, http://dx.doi.org/10.1145/2503009.
[45] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, I. Stoica, Tachyon: reliable, memory speed storage for cluster computing frameworks, in: Proceedings of the ACM Symposium on Cloud Computing – SOCC'14, 2014, pp. 1–15.
[46] R. Livingstone, The 7 Vs of Big Data. Insights. Rob Livingstone Advisory, 21 June 2013. Web. 27 Feb. 2016.
[47] LSST. Data Management. Data Management. N.p., 2015. Web. 24 Feb. 2016.
[48] M. Maciejewski, To do more, better, faster and more cheaply: using big data in public administration, Int. Rev. Adm. Sci. (2016).
[49] G. Malewicz, M. Austern, A. Bik, Pregel: a system for large-scale graph processing, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, 2010, pp. 135–145.
[50] B. Marr, Big Data: Using SMART Big Data, Analytics and Metrics to Make Better Decisions and Improve Performance, John Wiley & Sons, 2015.
[51] R.R. McCune, T. Weninger, G. Madey, Thinking like a vertex, ACM Comput. Surv. 48 (2) (2015) 1–39, http://dx.doi.org/10.1145/2818185.
[52] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, A. Talwalkar, et al., MLlib: machine learning in Apache Spark. CoRR. Retrieved from, http://arxiv.org/abs/1505.06807.
[53] C. Olston, B. Reed, U. Srivastava, R. Kumar, A. Tomkins, Pig Latin: a not-so-foreign language for data processing, in: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data – SIGMOD'08, 2008, p. 1099.
[54] Optimus Information. Understanding the 7 V's of Big Data. *Blog*. Optimus Information Inc, 18 Aug. 2015. Web. 27 Feb. 2016.
[55] S.A. Osher, Privacy, computers and the patriot act: the fourth amendment isn't dead, but no one will insure it, Fla. L. Rev. 54 (2002) 521.
[56] A. Pavlo, E. Paulson, A. Rasin, D.J. Abadi, D.J. DeWitt, S. Madden, M. Stonebraker, A comparison of approaches to large-scale data analysis, in: Proceedings of the 35th SIGMOD International Conference on Management of Data, 2009, pp. 165–178.
[57] I. Polato, R. Ré, A. Goldman, F. Kon, A comprehensive view of Hadoop research—a systematic literature review, J. Netw. Comput. Appl. 46 (2014) 1–25, http://dx.doi.org/10.1016/j.jnca.2014.07.022.
[58] G. Press, A very short history of big data. FORBES, 2013.

[59] W. Raghupathi, V. Raghupathi, Big data analytics in healthcare: promise and potential, Health Inf. Sci. Syst. 2 (2014) 3, http://dx.doi.org/10.1186/2047-2501-2-3.
[60] M. Van Rijmenam, Connecting Data and People. Datafloq Read RSS. Datafloq, 7 Aug. 2015. Web. 27 Feb. 2016.
[61] D.P. Rodgers, Improvements in multiprocessor system design, Comput. Archit. News 13 (3) (June 1985) 225–231.
[62] C.D.G. Romero, J.K.D. Barriga, J.I.R. Molano, Big data meaning in the architecture of IoT for smart cities, in: International Conference on Data Mining and Big Data, Springer International Publishing, June 2016, pp. 457–465.
[63] M. Saecker, V. Markl, Big data analytics on modern hardware architectures: a technology survey, in: Lecture Notes in Business Information Processing (LNBIP), vol. 138, 2013, pp. 125–149.
[64] S. Sakr, A. Liu, A.G. Fayoumi, The family of MapReduce and large-scale data processing systems, ACM Comput. Surv. 46 (1) (2013) 1–44, http://dx.doi.org/10.1145/2522968.2522979.
[65] M. Schwarzkopf, A. Konwinski, Omega: flexible, scalable schedulers for large compute clusters, in: EuroSys'13 Proceedings of the 8th ACM European Conference on Computer Systems, 2013, pp. 351–364.
[66] J. Shamsi, M.A. Khojaye, M.A. Qasmi, Data-intensive cloud computing: requirements, expectations, challenges, and solutions, J. Grid Comput. 11 (2) (2013) 281–310, http://dx.doi.org/10.1007/s10723-013-9255-6.
[67] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The Hadoop distributed file system, in: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST2010, 2010.
[68] D. Singh, C.K. Reddy, A survey on platforms for big data analytics, J. Big Data 2 (1) (2014) 1.
[69] A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, R. Murthy, et al., Hive – a petabyte scale data warehouse using Hadoop, in: Proceedings – International Conference on Data Engineering, 2010, pp. 996–1005.
[70] A. Toshniwal, J. Donham, N. Bhagat, S. Mittal, D. Ryaboy, S. Taneja, M. Fu, et al., Storm@twitter, in: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data – SIGMOD'14, 2014, pp. 147–156.
[71] C. Uzunkaya, T. Ensari, Y. Kavurucu, Hadoop ecosystem and its analysis on tweets, Proc., Soc. Behav. Sci. 195 (2015) 1890–1897.
[72] V.K. Vavilapalli, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, H. Shah, et al., Apache Hadoop YARN, in: Proceedings of the 4th Annual Symposium on Cloud Computing – SOCC'13, ACM Press, New York, New York, USA, 2013, pp. 1–16.
[73] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, J. Wilkes, Large-scale cluster management at Google with Borg, in: Proceedings of the Tenth European Conference on Computer Systems – EuroSys'15, 2015, pp. 1–17.
[74] M. Wang, G. Zhang, C. Zhang, J. Zhang, C. Li, An IoT-based appliance control system for smart homes, in: 2013 Fourth International Conference on Intelligent Control and Information Processing (ICICIP), IEEE, 2013, pp. 744–747.
[75] R.S. Xin, J.E. Gonzalez, M.J. Franklin, I. Stoica, GraphX: a resilient distributed graph system on spark, in: First International Workshop on Graph Data Management Experiences and Systems, 2013, p. 2.
[76] S.B. Yoginath, K.S. Perumalla, Design of a high-fidelity testing framework for secure electric grid control, in: Proceedings of the 2014 Winter Simulation Conference, IEEE Press, December 2014, pp. 3024–3035.
[77] M. Zaharia, M. Chowdhury, T. Das, A. Dave, Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing, in: NSDI'12 Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, 2012, p. 2.
[78] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, I. Stoica, Spark: cluster computing with working sets, in: HotCloud'10 Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, 2010, p. 10.
[79] M. Zaharia, T. Das, H. Li, S. Shenker, I. Stoica, Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters, in: Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Computing, 2012, p. 10.