University of Reading

Mapping the big data landscape: technologies, platforms and paradigms for real-time analytics of data streams

Article

Published Version

Creative Commons: Attribution 4.0 (CC-BY)

Open Access

Dubuc, Timothée, Stahl, Frederic ORCID logoORCID: https://orcid.org/0000-0002-4860-0203 and Roesch, Etienne B. ORCID logoORCID: https://orcid.org/0000-0002-8913-4173 (2021) Mapping the big data landscape: technologies, platforms and paradigms for real-time analytics of data streams. IEEE Access, 9. pp. 15351-15374. ISSN 2169-3536 doi: https://doi.org/10.1109/ACCESS.2020.3046132 Available at https://centaur.reading.ac.uk/95419/

It is advisable to refer to the publisher's version if you intend to cite from the work. See Guidance on citing.

To link to this article DOI: http://dx.doi.org/10.1109/ACCESS.2020.3046132

Publisher: IEEE

All outputs in CentAUR are protected by Intellectual Property Rights law, including copyright law. Copyright and IPR is retained by the creators or other copyright holders. Terms and conditions for use of this material are defined in the End User Agreement.

www.reading.ac.uk/centaur

University of Reading

CentAUR

Central Archive at the University of Reading

Reading's research outputs online

IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received November 23, 2020, accepted December 11, 2020, date of publication December 21, 2020,
date of current version January 27, 2021.

Digital Object Identifier 10.1109/ACCESS.2020.3046132

Mapping the Big Data Landscape: Technologies, Platforms and Paradigms for Real-Time Analytics of Data Streams

TIMOTHÉE DUBUC[1,2], FREDERIC STAHL[3], AND ETIENNE B. ROESCH[1,2]
[1]School of Psychology and Clinical Language Sciences, University of Reading, Reading RG6 6AH, U.K.
[2]Centre for Integrative Neuroscience and Neurodynamics, University of Reading, Reading RG6 6AH, U.K.
[3]Laboratory Niedersachsen, German Research Center for Artificial Intelligence GmbH (DFKI), 26129 Oldenburg, Germany

Corresponding author: Etienne B. Roesch (e.b.roesch@reading.ac.uk)

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC) through the European Coordinated Research on Long-Term Challenges in Information and Communication Sciences and Technologies ERA-NET (CHIST-ERA), through the Project COCOON, under Grant EP/P016448.

ABSTRACT The 'Big Data' of yesterday is the 'data' of today. As technology progresses, new challenges arise and new solutions are developed. Due to the emergence of Internet of Things applications within the last decade, the field of Data Mining has been faced with the challenge of processing and analysing data streams in real-time, and under high data throughput conditions. This is often referred to as the Velocity aspect of Big Data. Whereas there are numerous reviews on Data Stream Mining techniques and applications, there is very little work surveying Data Stream processing paradigms and associated technologies, from data collection through to pre-processing and feature processing, from the perspective of the user, not that of the service provider. In this article, we evaluate a particular type of solution, which focuses on streaming data, and processing pipelines that permit online analysis of data streams that cannot be stored as-is on the computing platform. We review foundational computational concepts such as distributed computation, fault-tolerant computing, and computational paradigms/architectures. We then review the available technological solutions, and applications that pertain to data stream mining as case studies of these theoretical concepts. We conclude with a discussion of the field of data stream processing/analytics, future directions and research challenges.

INDEX TERMS Big data applications, Internet of Things (IoT), edge computing, distributed computing, pipeline processing.

I. INTRODUCTION

Stemming from recent technological advancement, what came to be coined the 'Data Era' [1]–[3] is now concurrent to a dramatic increase in the portability of computerized devices. The omnipresent inter-connectivity of these technologies supports access to an unprecedented amount of information and, more so, to the sources of this data in real-time [4]. Nearly every action on an electronic device generates data, which is stored for future use because of its foreseen informational value.

Raw Data, however, is now almost rendered useless as their unrefined and unconstrained nature, combined with the amount of redundancy and sheer volume make them difficult to manage [3]. In order to benefit from this data, there is thus a growing need for data contextualisation and interpretation; until the end of the last decade, this work was conducted by human experts and data was gathered at an increasing pace such that we are currently generating more data than any human could ever assimilate [5], [6].

Big Data analysis techniques, i.e. the method of processing and analyzing 'Big' amounts of data, is proposed as the solution. Through the use of computerized frameworks and architectures, human experts may augment their abilities, rendering the distillation of raw data and the subsequent decision-making automation tractable.

The associate editor coordinating the review of this manuscript and approving it for publication was Jing Bi.

A. BIG DATA, OVER 20 YEARS LATER

The term Big Data is commonly thought to have been coined by John R. Mashey in 1998 at Silicon Graphics [7], [8]. However, its true origin may extend back to the early 90s [9]. Since then, its popularity has grown in parallel with the exponential increase of the amount of data being produced,

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

to the point where, in the present day, everyone in the IT world wants 'to do Big Data' [2], [5], [10]–[15].

The term Big Data suffers from a 'buzz word' effect, which led to widespread imprecise definitions, however it is typically understood as an umbrella term encompassing a variety of techniques applied to the analysis of large amounts of data within a defined timeframe [12], [16], [17] (See Figure 8). This concept of Big Data poses sizeable multidisciplinary challenges, including the need for distributed and high-performance computing, parallel computing, statistics, machine learning, non-structured data treatment, networking, database design, and IT infrastructure. In light of the lack of a precise and consensual definition [18]–[20], for the purpose of this article, we consider 'Big Data processing' to include *any operations conducted on a dataset resulting in a process too voluminous to be handled efficiently and in a tractable way on a single machine*. This is consistent with a previously used definition [21], and can be extended to include data acquisition, management, and storage: all of which are aspects that became critical to the way data are processed, often in server-less contexts.

Despite these semantic considerations, this new field has had a striking impact on industries and institutions, who suddenly realized the value of the data they had been collecting without a clear purpose [14]. The thorough analysis of that information enabled many organisations to optimize their business strategies and more efficiently target potential opportunities [2], [6]. This trend also led to the emergence of 'Big Data' consultancies that offered support and services to companies that wanted to use 'Big Data' but did not have the appropriate expertise or resources to do so. [15], [22]–[24].

As established companies navigate this new and growing technological landscape, the number of industry-specific issues grows alongside the number of technological solutions available, and this upward trend leaves newcomers with the difficult task of understanding and assessing the available Big Data tools based on their and characteristics.

B. THE SOLUTION LANDSCAPE

Understanding the architectures available to 'do' Big Data is no simple task: of the numerous solutions in the market, no single problem comes with a unique middleware combination that will solve the issue; there is no generic way to do things. In addition to the difficulty arising from the sheer number of tools at our disposal, other factors complicate and influence decision making.

Big Data empowering businesses, termed 'providers', position themselves as intermediary agents facilitating enterprise operations, either through a service and/or the distribution of ready-to-use tailored solutions. In the former case, the provider will disclose some documentation and APIs for clients to make use of the service provided, billing the service accordingly and retaining the information about the code and/or any optimized setup of the core service behind closed doors (the 'black box' business model). In the latter case, the provider often provides the solution for free (sometimes open source) with minimal developer/administrator documentation, adopting a business model centered on monetisation of customer assistance (the 'grey box' business model).

Unfortunately, many solutions offer only slight variations to each other within the same processing paradigm, and are most often based on outdated scientific publications with limited relevance to the state of the art. As they are initially created to answer a specific problem, these solutions have their own innovations and operative modes (Apache Kafka/Samza by LinkedIn, OpenStack by Rackspace Hosting and NASA, Apache FlumeJava/Millwheel/Beam by Google, etc), but as they are developed further, they may be extended to operate outside their original specifications, regardless of whether they can excel and respond to the specific needs of their creators in their modified state.

This variability creates a fragmented technological landscape that is difficult, if not impossible, to map using a unified, quantified benchmark. Although attempts at mapping the landscape have been made [25]–[27], the usefulness of these works suffers from the small number of systems they can address, compared with the large number of potential system combinations: a given benchmark is only valid where it is performed on comparable hardware, using the exact same version of the code and dependencies -constraints that can rarely be met when reproducing published benchmarks.

Note that a substantial effort has been made to formulate hardware-agnostic benchmarking tools capable of bypassing these limitations in order to compare solutions within their own contexts. Some are expressed as a platform-specific setup [28], [29], and others as comparison hubs hosting competitions on real-life synthetic problems. Unfortunately these approaches typically require substantial R&D efforts, which typical users would not engage with.

In the present review, we provide a field guide for beginner Big Data consumers. Due to the practical limitations of benchmarking, we approach this question from a qualitative perspective to produce a lasting frame of reference for the variety of theoretical paradigms and options available. We achieve this, first, by reviewing the theoretical concepts surrounding the characterisation of the data and the infrastructure supporting their transformation, before handpicking the most popular middleware as mediums to discuss the implementation of these theoretical paradigms and possibilities. Through this approach, we hope to provide both technological pointers and practical explanations of the different technologies powering the IT backbone of many industries that leverage Big Data. Finally, we discuss the technologies' relevance through the lens of practical and hypothetical use cases, discussing (1) the types of applications available, (2) the tools best suited to address the applications, and (3) the problems that could potentially arise in the future considering the technological trends we observe today.

II. COMPUTATIONAL CONCEPTS

We introduced Big Data processing as being an interdisciplinary enterprise. The reasons for this stem from the

requirements for handling big volumes of data with minimal processing time and high reliability, which force the development of specific architectures that can cope with high workloads and scale efficiently. Here we introduce the concepts that define those requirements, and discuss the nature of the data (and its metrics), the concepts of task distribution (and scheduling), fault tolerance, and data availability. Finally, we review the three main computational architectures that form the foundations of the different engines presented in Section III.

A. DATA AND DATA STREAMS

The data processed and stored in a given system can be characterised using metrics, regardless of its intended application. These metrics were initially referred to as the '3D/3Vs' [30] and later, as the *'4Vs'* of Big Data [31]–[33] are the: Volume, Velocity, Variety and Value/Veracity of the data [34], [35] and each pertains to a particular aspect of information.

Data **Volume** represents the order of magnitude of the size of the data without concern for its encoding or its location.

Data **Velocity** characterises the rate at which a certain volume of data move. This adds the dimension of time to the data volumes in the form of frequency of occurrence, arrival, and the speed of data transfer, throughput, and bandwidth [23], [36]. Velocity is becoming more relevant and critical as cloud services are now omnipresent, thus these services are accessible by anyone, anywhere, at any time. This constraint, and the fact that those systems are dealing with human users, has drastic consequences in terms of data rate fluctuation as they will directly correlate with our human habits and limitations - and the type of habit they are designed to deal with.

All data are of a given type. Data **Variety** is a consequence of the technological developments of the last decades that saw the increase of the number of sources of data [37], [38]. This resulted in the discovery, creation and storage of unstructured data: less artificially crafted segments of information that were not necessarily suitable for relational database models. Unstructured data often complement structured data and can originate from any sensor interfacing with the natural world (bio-sensors, scanners, camera, etc…), and in that respect, 'Big Data' storage and processing facilities are likely have to encounter such a paradigm and must be capable of handling it.

Data **Value** has direct consequences for the analysis processes in place, and denotes the informational worth of the data within a defined context [39]. This can range from user usage statistics providing useful market insight to payment details and/or financial data that form the backbone of an industry.

To that effect, we can add another V, the **Veracity** as indicator of the reliability of the data: decision making is one of the key aspects impacted by Big Data processing [39], and making decisions on reliable data known is imperative - as such, Veracity is a Value in itself.

Note that the meaning attributed to the different *Vs* is subject to variations (Visualisation, Value, Vulnerability), that are largely depend upon the context in which 'Big Data' is discussed, and on the level of abstraction required with respect to the technical constraint involved. These characteristics are important when considering or designing a system and its proposed usage. For instance, when dealing with data displaying high Volume but low Velocity, this often translates into a mitigated emphasis on low latency (see later in the text) as it becomes more important to maintain the integrity of a large amount of data when a batch is inbound, than to parse it quickly to free the system for the next batch. The Value of the data will also shape the importance of fault tolerance, whereby the more important the data, the more stable and robust the system must be, and thus redundancy often becomes an asset.

Finally, the Veracity of the data will directly impact the requirements for the manipulation, storage and processing of the data [39], and each of these aspects may alter the meaning of the information thus need to be carefully considered to prevent any data corruption. These steps are crucial for the efficiency and exactness of any subsequent decision making process based on the collected data.

Dimensionality: An element paramount to this context is the minimum number of dimensions necessary to describe the data to be stored (data dimensionality). Typically, this does not affect the choice of the storage hardware, or the overall architecture of the service, however, some database types are better equipped to deal with certain types of data [23], and some hardware have advantages over others when it comes to highly parallel workloads [40].

As an example, consider the following types of dimensionality:

- Uni-dimensional: holds a constant value without a time dimension to store its evolution (e.g. user ID or password)
- Bi-dimensional: intensity varying through time (e.g. monochannel sound)
- Tri-dimensional: x,y location components and a pixel value of a grayscale immage
- Quadri-dimensional: x,y locations of components, pixel value and time (grayscale video)

As the number of dimension increases linearly, the volume of data, expressed in raw bits, increases exponentially. This calls for low complexity algorithm, dimensionality reduction techniques, and specialized hardware (such as GPU, TPU, FPGA…) to support the processing of large amounts of data [41], [42].

Latency and Throughput: Another important element is the delay in data processing and analytics relative to the data access in memory, and the time taken by a system to deal with this. When data are being streamed, the time it takes to upload the data into the system is a major factor. Latency characterises the reaction time of a system, e.g. the response time of a website when a user interacts with it, or the time

needed for a Big Data application to access and retrieve the data to operate on [42]. The less latency a system incurs, the more reactive it is. When involved in a computational system, this translates into less idle time during which your data are simply waiting for the engine/system to be able to acknowledge them. Typically, for batch systems, the latency time is expressed in hours, whereas real-time systems aim to express latency in milliseconds, and employ caching mechanisms. Another element of data handling systems is the amount of data they can manage per unit of time, i.e. the throughput. Unlike latency which is bounded to the software design, throughput is mainly determined by the nature of the task and the hardware supporting the operation. Note, also, that latency is not directly linked to throughput: a system with high latency can display a high throughput but will lag behind on user-driven events (batch processing). More details on this topic can be found in [17].

Batch versus Stream of Data: Independently to the *4Vs* of Big Data is the structure of the assimilation algorithms employed to process the data: either 'as they come', also termed *Stream* processing, or grouped into chunks on demand, termed *Batch* processing.

Stream processing is more similar to event driven systems where the environment drives their reaction as activation occurs; they generally enjoy low latency, capturing interaction between different events unfolding and feel very natural to integrate within user facing systems.

Batch processing is a direct consequence of the specificity of the hardware supporting the algorithm: within a traditional x86/x64 architecture, physical processing units and memory are separated, creating latency when a set of data has to be requested from either the long term storage (HDD, SSD) or the volatile memory (RAM). Therefore, instead of doing multiple data requests and summing the latency of each call to calculate the overall completion time, it is more time efficient to wait for all the data requests to be known and bundle them into a single memory access. This is even more true when GPU based computing is considered, as the video memory (VRAM) is generally separated from the central system memory, and in this case, the addition of all data request latency could quickly outweigh any benefit of using GPU computing over CPU computing.

In previous sections, we reviewed the properties that characterise the data and their delivery to a given system, however we presented the properties irrespectively of the type of application or context-specific constraints. Those context-specific constraints are the factors separating an ideal 'do-it-all' system from the systems available in reality. Notably, they result in the following common issues and practices:

- **High volume computing comes with the constraint of tractability**: The goal of conducting an analysis is to create an informative conditioning of a given dataset. This should, therefore, be done in tractable time and within the time window the data remain relevant: e.g., there is little practical interest in predicting the price for crude oil 20 years ago. This variety of challenge begins at the data ingestion stage - reading data from a file of several hundred MBs up to several GBs is common, however, attempting to read data from a file of TBs or even PB size becomes very challenging. For such an amount of data, the design of the system handling the communication between the data source and the different parts of the analytic pipeline can create a major bottleneck. In addition to needing the right choice of software/hardware architecture, routine optimisation and partial processing (only selecting the fraction of data representative of the whole) are common workarounds [43].
- **When we cannot process any faster, we try to process many things at once**: A single computational node, even when using multiple cores (*parallel computation*) can only process as fast as its hardware enables; even considering the right data subset to process and the optimal algorithmic implementation, this threshold would constitute a theoretical upper limit. Common practice is to set up many parallel physical processing units capable of digesting data faster than a single unit could (*distributed computation*) but this approach also has its limits, as not all problems may be broken down into tasks that can occur in parallel.
- **Parallelism comes with its own formalism and costs**: When a problem is to be allocated to a group of machines (*mapping*, a focus of some paradigms), the subsequent individual tasks have to be coordinated (*scheduling*) and distributed such that no single node would choke under the workload while others stood idle (*load balancing*). Results have to be collected and conditioned (*reduced*, a focus of some paradigms) for storage. Those two apparently simple steps come with a susceptibility to failure and numerous technical constraints. Distributing work across multiple machines relies heavily on the network connecting the nodes, whereby if a link, or the recipient of a task fails at a critical moment, the system can be left in an undetermined state potentially leading to data being lost or duplicated. Therefore, the constant monitoring of the system is required to assure tasks occur successfully, which is not always feasible. Finally, it is worth mentioning that distributed computation comes with an overhead that stems from the necessity to pack, transmit, and unpack the data when transferred from a node to another. As a consequence, many machines working together on a single problem will never be able to solve as many tasks as the sum of their computational power would lead us to expect.
- **Large jobs should not be lost**: Even when using a computational cluster, some analyses can take days, weeks or longer. Upon carrying out such an enterprise, it is important to ensure the completeness and exactness of the results. As such, technical failures should be handled in such a way that their occurrence would not prevent completion (*fault tolerance*) of the entire process

nor corrupt the data (lost, duplication, undetermined state, etc…). This calls for mechanisms such as queuing systems, message re-delivery, task re-scheduling, and/or result check-pointing that although necessary, will inevitably generate additional overheads.

The above description of common problems faced by data processing systems provides insights into what an ideal architecture would look like: a system capable of managing a large volume of tasks in parallel, in a resilient environment, that is easy to monitor and manipulate, and that scales well. Such a system is (to date) purely hypothetical, however, distributed systems present numerous qualities similar to this ideal system.

Within distributed systems, the reliance on many machines removes single points of failure from the system, rendering it more resilient in the long term. It also becomes possible to maintain and upgrade the system without any service disruption, by unplugging one/more nodes at a time and re-routing resources. Moreover, this architecture enables systems that scale out rather than (or in addition to) scale up: namely, the system's performances and capabilities will grow according to the number of nodes joining the existing pool, whether hosted in the same space or rented out from elsewhere. This, theoretically, enables larger growth than a 'scaling up' system as in this scenario, we are not constrained by the physical limit of the hardware. The power of the resulting architecture is, however, only matched to the complexity involved: the addition of middleware layers and services can result in 'bulky' systems that are too difficult to adapt to specific needs. In the following subsection, we discuss a number of aspects that must be considered when designing such a platform.

B. TASKS DISTRIBUTION AND COMPUTATION SCHEDULING

Task distribution is an important component of both distributed and 'Big Data' processing, as it prevents the under-exploitation and/or exhaustion of resources. If at any point, the main work queue is populated, all of the computational nodes should be filled to capacity in order to minimise idle time and resource loss, while respecting the relative urgency of the different tasks and the minimum Service Level Agreement expected from the system. Within the well-adopted Apache Hadoop, this is achieved through a separated service (Apache YARN - see Section III) that manages the dynamic negotiation of the resources. The priority of such a software is to reduce latency and network congestion by distributing the task with respect to node capacity and data location. The system should prioritize tasks performed locally, followed by those within the same vicinity (rack), rather than using nodes across the board within the same data center.

Although a multitude of schedulers exist, only a few paradigms tend to be used; these will not be discussed in full here, but have previously been reviewed by Etsion *et al.*, and Sliwo *et al.* [44], [45]. Here, we present examples of

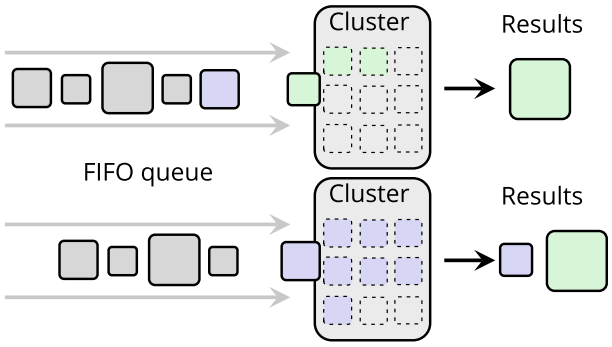

FIGURE 1. FIFO scheduler representation: A FIFO Scheduler handles tasks as they arrive, without concern for the requirements in terms of computing units. The top part of the figure exposes a task that only requires two compute units out of nine. The following task (that requires seven compute units - bottom part of the figure) will nonetheless have to wait for the completion of the previously submitted small task.

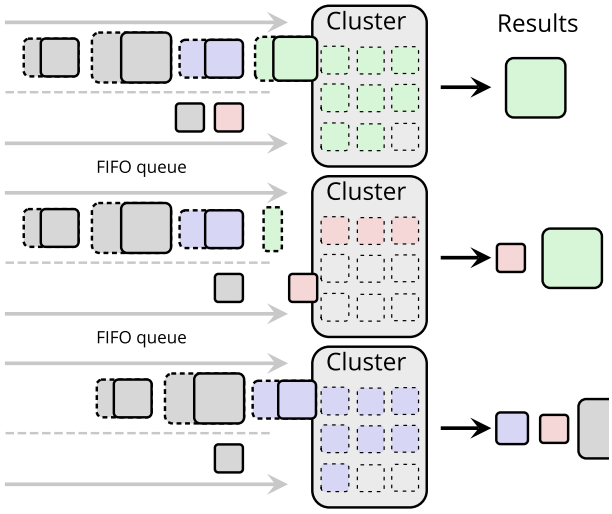

FIGURE 2. Maui scheduler representation: A Maui Scheduler builds statistics relative to task executions and estimates how much time will be needed to carry out a computation. The next task will, then, be planned accordingly. In the figure, the top pipeline shows the green task currently being executed and estimated to take longer than it will actually take to complete (extending dashed line the left part of the green task). Should a task finish ahead of schedule (middle part of the figure), a small task parked in a backfill queue will be inserted as padding prior to the next scheduled task. This effectively reduces the overall turnaround time and increases the throughput of the cluster.

schedulers that we believe are representative of the most important aspects of large scale computing:

FIFO/FCFS Scheduler. This paradigm is one of the simplest scheduling frameworks possible. The assumption is that tasks should be executed in order (First In, First Out or First Come, First Served) [46], even if this may be inefficient overall (See Figure 1). While the simplicity of such an arrangement results in a very low overhead, and in the absence of resource starvation (no process interruption or computational reduction), there are drawbacks. For example, the computational resources can be held in a locked state by a long lasting process and/or resource that is temporarily inaccessible;

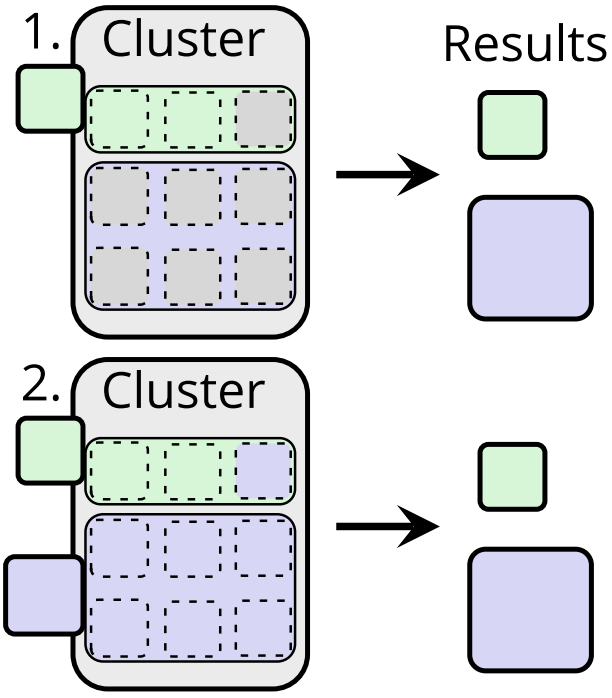

FIGURE 3. Capacity scheduler representation: A Capacity Scheduler provisions resources ahead of time and attributes them to different entities. Here, the cluster has been split into two blocks of three and six compute units (between entity green and entity blue), respectively. Should a given task not use all the resource allocated (1), the scheduler will allow the larger task to overflow and re-allocate spare resource to bigger tasks that can use it (2). Regardless, each entity holds priority over its quota and the overflow permission will be revoked, should a more demanding task be posted by the green entity.

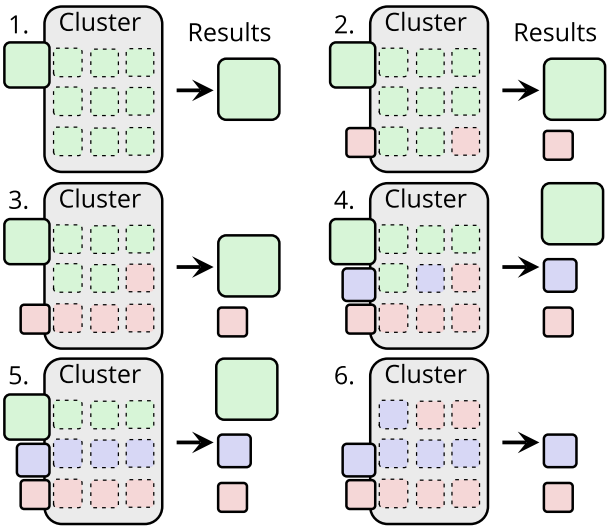

FIGURE 4. Fair scheduler representation: A Fair Scheduler will always endeavour to allocate the resources evenly among the submitted tasks. (1.) shows a single task on the cluster running on all compute units until (2.) a new task is submitted. The cluster will progressively re-distribute the resources available until an even division is reached (3.). The process also occurs should a task complete (4., 5. and 6).

scenarios that are likely to increase the turnaround time of other jobs significantly, regardless of their respective size or priority.

Maui Scheduler. Widely used in the early 2000s among the HPC community, this batch scheduler aims to order the execution of tasks to minimize turnaround time [47]. This is achieved through its advanced reservation and backfill scheduling capabilities: the Maui Scheduler estimates the users' execution time for a job by calculating statistics on the tasks involved. This information, alongside the stated priority of the task, is used to reserve a time window during which the job will run. The execution schedule queue is filled following a priority FIFO queue, and in accordance with the required amount of resources available or predicted to be available, at a given time. Subsequently, a backfill phase fills the scheduling gap with jobs of lower priority and/or smaller size (See Figure 2). This has been shown to increase the usage of HPC centers by 20% compared with scheduling systems without backfill mechanisms. This set up is not flawless as efficiency increases are typically perceptible for small tasks, whereas medium or large tasks will gain no benefit from the backfill mechanism as they are too big to fill gaps between tasks of higher priority.

Capacity Scheduler. This scheduler permits the sharing of a single (Hadoop) cluster among many entities in such a way that the different applications are allocated resources in a timely manner, and taking into account hardware capacity (See Figure 3). The practice of sharing a cluster is a direct consequence of the high cost of maintaining such an infrastructure: not every organization can afford the investment and for those who can, under-exploitation of the cluster constitutes resource wastage. Furthermore, any cluster of a significant size have periods of high stress (peak of demand) and idle periods, during which a high cost infrastructure is being under-utilized. The Capacity Scheduler is designed to allow each user/entity to be allocated a capacity quota, with the added possibility to use any excess resources not currently allocated to a task [48]. This provides high throughput and elasticity, maximizing cost-efficiency. Those features come with capacity/time safeguards preventing a malicious or excessive allocation of resources. The overall capacity splitting and allocation is governed by a set of isolated queues, linked to a set of computational resources, through which entities (i.e. users) can submit jobs with user-defined priority levels.

Fair Scheduler. Unlike the Capacity Scheduler, the Fair Scheduler does not split the computer cluster (i.e. a Hadoop cluster) into predefined computational blocks, and instead aims to ensure that any job on the cluster is allocated an equal share of the resources [49] (See Figure 4). In this configuration, if a single task is running, it occupies the entire cluster. If a second task was submitted, the resources initially occupied by the first task would be slowly reallocated, to the new task until a 'fair balance' is restored. The partitioning of the cluster can be shaped by the priority of the task to be executed, by altering the weights used to determine the fraction

of computation time allocated to every job. Jobs are organized into pools (owned by individuals or groups), each benefitting from a fair share of the computational resources; each pool supports fair sharing or FIFO scheduling of the tasks submitted, and is guaranteed a minimal level of resources upon submission of a job. If this minimal operational criterion cannot be met (all nodes are already allocated), the pool can be configured to support killing tasks from other pools, thereby ensuring the availability of resources. A key advantage of this configuration is that killing an operation does not result in the loss of a job, as computational tasks are fault-tolerant and will be rescheduled (as it is implemented in Hadoop).

(HT)Condor. Created in 1988, Condor (renamed HTCondor in 2012) was introduced as a distributed scheduler capable of managing a dedicated Beowulf cluster (see Section II-E1). No centralized submission point exists within the system, and every node hosts its own job queue. It also includes a meta-scheduler accepting acyclic task graphs by making the successful completion of a set of tasks a dependency for future jobs. Moreover, it offers a functionality that many do not: it combines the latent computational power of idle workstations to complement the resources of a dedicated cluster. This feature is extended by 'ClassAd' which enables the matching of specific 'resource requests' with specific 'resource offers', thus enabling both jobs and machines to specify their preferences, in term of running platforms and appropriated tasks. For more information about this advanced scheduling platform see Tannenbaum 2001 [50], or the HTCondor website [51].

C. FAULT TOLERANCE

Fault tolerance is a central element of 'Big Data' computation - even the fastest systems will eventually encounter a task requiring hours to complete, yielding vulnerability to ongoing tasks. The loss of a running task might not seem like a big problem, however, should the task need to be completely restarted, this can create a substantial setback to the overall processing pipeline; this would in turn create financial costs, which can rapidly multiply. It is therefore imperative to preserve the integrity of running tasks. In this section, we review the features of fault tolerance that current solutions attempt to mitigate, before reviewing some of those solutions (see Section III).

1) DATA PRESERVATION

To prevent loss of data, multiple policies can be used to regulate storage: (1) task sensitive data can be duplicated, preferably on separate hardware, to avoid a cascading failure, or (2) fault-tolerant file systems can be used to mitigate hardware and/or system failure.

HDFS (Hadoop) [52], Cosmos (Microsoft) [53] or GPFS (IBM) [54] are examples of file systems suited to this task. Alternatively, including a solution involving Redundant Arrays of Inexpensive Disks (RAID) within the hardware/software architecture could increase the possibility of recovering from a storage failure, in addition to boosting the I/O relative performances (RAID 5/6).

Some of these solutions constitute distributed data storage, and therefore fall under Brewer's conjecture that such a medium can (at best) provide only two of the following three guarantees [55]:

- 'Consistency': the ability to deliver the latest version of a data item, according to the latest modification.
- 'Availability': the ability to deliver some data in the event of a request.
- 'Partition tolerance': the ability to continue operating in the event of degradation of the network quality (packet loss, delay, network partition, ...).

Under normal conditions, a system can operate without compromising in terms of Consistency and Availability. The choice only arises in the event of a network failure, if the system were to remain operational. In 2002, this exclusive choice was proven mandatory by Seth Gilbert and is now known as the 'CAP Theorem'.

2) MESSAGE DELIVERY

Message delivery drives the reliability and performance of a distributed system. The multiple processes involved must be able to communicate, coordinate, and transfer data as needed. However, reliance on communication represents a weak spot within the architecture, as it relies on the quality of the network and on a larger scale, the 'reachability' of the nodes. Imperfect by nature, this calls for a set of mechanisms to detect a system failure and respond accordingly; three paradigms are currently used within the solution landscape:

- At-most-once delivery (avoid duplication) [56]
- At-least-once delivery (for completeness concerns) [57]
- exactly-once (in an ideal world) [58]

Generally, all paradigms implement a check-pointing system that asserts the correct departure of a message, its receipt, and the existence of a functioning link between two nodes. Consider two nodes communicating (a master and a slave) and transmitting a unique message - how many messages will be delivered if everything works correctly? If one packet is lost? If the slave crashes? If the master crashes? If a combination of failures occurs?

Naive communication involves the blind dispatch of the message from the master to the slave. In the event of a failure, the packet is lost during transmission, but no duplicate is possible (i.e. at-most-once delivery systems). The simplest fault-tolerant communication protocol involves requiring an acknowledgement from the slave node upon receipt of the message. In such a configuration, a lost packet will trigger the re-submission of the message after an established period of time. However, if the network fails during the acknowledgement phase, this will instead trigger the generation and receipt of a duplicate message (i.e. at-least-once delivery systems). Preferably, a system should be able to deliver a message,

regardless of the condition, once and only once. Such a solution requires the presence of a counter on each node, that identifies the origin and the unique ID of each transaction. In the event of a failure, if re-delivery occurs, a duplicate would be detected as the transaction's unique ID would be present twice [59]. This is a very simplified vision of what an exactly-once semantic system could be, but the actual implementation of such a system would have more technical problems than can be addressed in this review. However, it remains that the tracking of transaction states holds a central place in the mechanisms enabling this functionality, and with that, the preservation of the different tasks running.

3) TASK PRESERVATION

The end-goal of all fault-tolerance mechanisms is to maintain the operation of the system until all tasks are completed successfully, and the data are securely stored. When it comes to task completion, the first and perhaps most common factor that could put the operation in jeopardy is human error. In a well-designed environment, if a job sub-task ran a faulty code implementation or met a limit-case, this should not impact the rest of the system. To ensure this, most compute engines start each worker in separated context (multiple JVM process, linux cgroups, virtual machines, containers, etc…) [60]–[63] enabling a process to die without affecting the overall system.

In the case of a hardware failure, the re-scheduling of the operation should be ensured to prevent a task from disappearing; this is generally handled by the scheduler in charge of monitoring the health of each task. Other engines keep a log of the operations performed to attempt recovery of context and data in the event of a system failure.

Finally, one of the most sensitive features related to the robustness of a system is the presence of 'single points of failure': a unique part or function upon which depends the operation of the whole system. Should a central node (typically a control node) or process cease to function, the entire system would become inoperative and stop in an undetermined state, leading to extensive maintenance and downtime. Most modern systems offer the possibility to be distributed/duplicated over multiple identical nodes (redundancy) to prevent such weaknesses.

D. DATA AVAILABILITY

Data availability is a substantial concern when dealing with large amounts of data. If handled incorrectly, it can have immediate adverse consequences on system latency and throughput - as each computational task tries to acquire the data it needs, it will also need to retain control of the computational resources required for the task, for however long data retrieval takes.

This problem can extend further: in the event of a faulty node or unstable network, data retrieval may need to be repeated several times until a new node takes over the task of the faulty/unreachable node. This can be catastrophic if communication fails at a critical moment (i.e. peak usage), and the consequent recovery procedures trigger cascading issues that impact system latency.

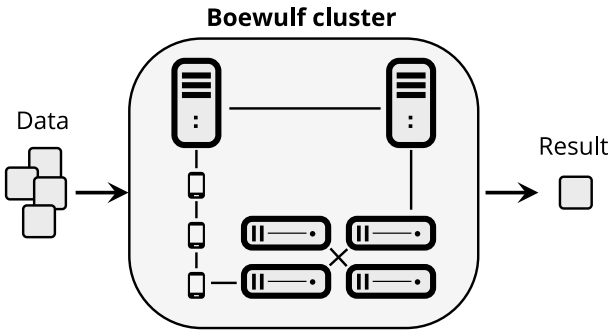

FIGURE 5. Beowulf cluster-based computation: A Beowulf cluster composed of multiple pieces of convenience hardware ingests multiple data entries. The computation scheduling and data routing are handled by the custom software running along with the data. No built-in fault tolerance exist. The result can be retrieved whenever the code execution completes.

Mitigating measures are typically taken to decrease the danger of hardware failure but are not aimed at decreasing the system's data-driven latency. For this, at least one version of the data should be located close to the compute node and, depending on the problem, should persist in a shared memory [64]. Note that this mainly applies to recursive tasks acting multiple times upon the same data samples.

E. COMPUTE PARADIGMS

Thus far, we have introduced the concepts of distributed computing, and Big Data appliances as a pool of computational power. The reality is largely different from the theory however, as hardware and software constraints often structure the application pipeline. We now introduce four successful computational architectures that are used in various production environments; these paradigms are often tailored to suit particular markets, thus their individual implementation can differ.

1) BEOWULF CLUSTER

This paradigm is widely used when prototyping systems, including those that leverage multiple convenience computers. Originally tailored for a designated computer hardware (built in 1994 at NASA, by Thomas Sterling and Donald Becker [65]), today it does not require specific hardware. Typically, a Beowulf cluster is anything with computational power (See Figure 5), physically and logically interconnected in a way that allows a task to be distributed (LAN + software).

There are a number of good practices to follow (state saving, thread-safe data structures, communication synchronization, etc.), but it is important to remember that this architecture (unlike the Lambda or Kappa architectures, reviewed below), requires significant customisation. Typically, a Beowulf application setup involves some form of network abstraction (e.g. Parallel Virtual Machine [66]) to enable the easy parallelisation of tasks by creating a logical virtual machine using all available nodes, or fast inter-machine communication (e.g. InfiniBand, Message Passing

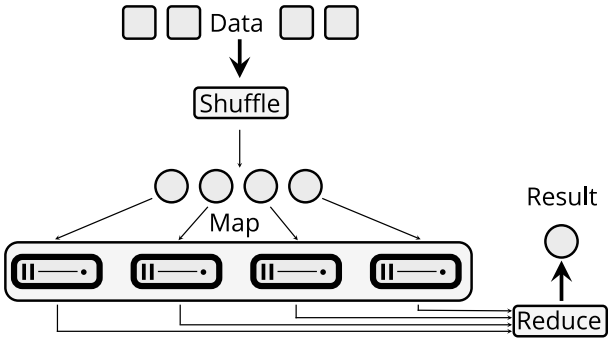

FIGURE 6. MapReduce based computation: A MapReduce computation paradigm composed of four computing units. The data are grouped in batch, preprocessed (Shuffle operation) and Mapped to the different compute units. Each data entry so dispatched is expected to be of equal size. The result is obtained upon completion of all individual computations by Reducing the multiple outputs into a single result.

Interface [67], [68]) to manage communication of heterogeneous nodes.

In essence, fault tolerance and data access in an application is the responsibility of the cluster Designer, unlike for other solutions that instead rely on the quality and performance of off-the-shelf software solutions. Although hosting a Beowulf cluster carries significant overhead, in terms of maintenance, this approach still offers advantages due to its tailored infrastructure that can optimise for low latency, communication, high memory performance, or distributed processing.

2) MapReduce

The MapReduce programming model is a distributed, fault-tolerant paradigm [69] where computations unfold in three parallel phases: (1) the **Map** method allows the initial transformation (sorting, splitting, normalization, ...) of the data, (2) the **Shuffle** operation reroutes the interdependent datasets toward the designated computation units, and (3) the **Reduce** step aggregates and consolidates the results [69] (See Figure 6). In a way, this paradigm can be considered a 'divide-and-conquer' strategy as the Map primitive allows the generation of individual datasets much smaller than the initial tuples, thereby increasing the tractability of the individual tasks to be undertaken (supposing that the initial task can indeed be parallelized).

The native resilience and scalability of the MapReduce paradigm is its main difference compared with Beowulf: by constraining the manner in which the code unfolds ($Map \rightarrow Shuffle \rightarrow Reduce$), it becomes possible to identify critical points supporting the computation and optimize/protect those points accordingly. Depending on the hardware specification of the nodes supporting the code, this approach is capable of dealing with large scale data (petabytes) within a reasonable time frame [69].

3) LAMBDA ARCHITECTURE

The Lambda architecture can be considered the pinnacle of Big Data computing: it is robust, well established [70], tested [12] and somewhat bulky [71] (See Figure 7). Lambda architecture can be divided into three main modules:

- A *batch processing layer* controls handling the heavy load of the computation. This MapReduce layer runs requests in bulk, distributing tasks over many cores/nodes (map operation), and aggregates the results upon completion (reduce operation) without latency related restrictions; in certain applications, tasks handled this way can take between hours and weeks to complete. Despite long computation times, this method remains one of the most efficient, and is a direct consequence of the architecture of currently available hardware solutions.
- A *Speed processing layer* controls balancing the high latency produced by the batch processing layer. This layer conducts incremental computation to provide partial results, thus enabling any service relying on its architecture to continue operating while waiting for the batch computation to complete. Through this pipeline, the overall system is able to handle applications with millisecond latency.
- A *Serving layer* ensures the results are coherent, as the Lambda architecture contains two data production pipelines that need to be merged correctly. This layer takes the most complete and up-to-date version of the result(s) to serve to the client application, efficiently intertwining the two time scales in an (almost) seamless way.

Within the treatment pipelines of Lambda, the operations are carried on immutable data - only addition and deletion operations are authorised. This paradigm is described as 'human fault-tolerance' by Nathan Marz and James Warren [72]: if some records imputed are invalid, their deletion and the re-computation of the results during the next batch will correct the overall data and metrics. This architecture comes with a set of characteristic features:

First, when erroneous data are entered and corrected, an entire batch of data has to be recomputed, which can take a substantial amount of time. However, as Lambda architecture acts on immutable, discretized data-sets, recomputing the data can be straightforward.

Second, each algorithm has to be implemented twice (a batch version and a real-time version) thereby increasing maintenance and development cost. Note that some high level programming frameworks (Summingbird [73]) solve the problem to an extent, by making it possible to implement the algorithm once, before compiling it for both paradigms at once.

Third, the architecture is not fit for every application: machine learning e.g., requires a large amount of iterative operations on which I/O times can have a huge impact. This is primarily true for Apache Hadoop, and some mechanisms have been implemented to by-pass it (see sub-section III-A1).

Finally, the requirement for two layers following two different paradigms often means that two different processing

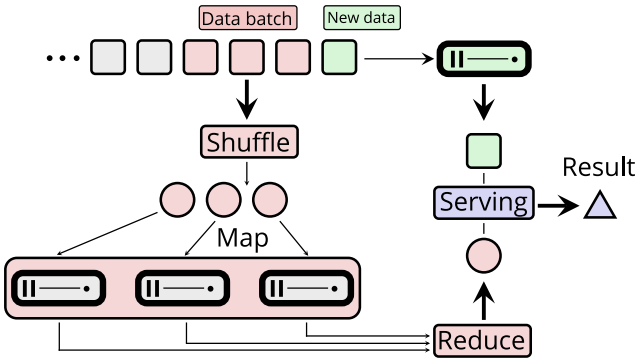

FIGURE 7. Lambda architecture driven computation: A Lambda architecture computation pipeline composed of two main parts: a MapReduce, batch processing, pipeline (in red) and a Speed processing layer (in green). As data are submitted over time, they are grouped in batch and submitted to the slow MapReduce pipeline. Each new data entry added during the batch processing life cycle is handled by the Speed processing layer. This will incrementally update a temporary cheap approximation of the result space. The serving layer (in blue) aggregates both, the exact output of the slow Batch layer and the fast approximation of the Speed processing layer, always serving the best-known result for the timestamp desired.

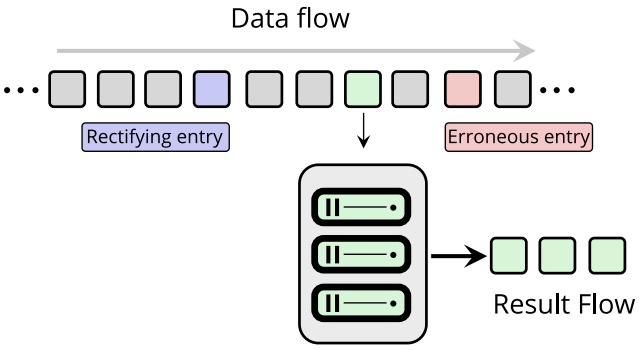

FIGURE 8. Kappa architecture driven computation: A Kappa architecture computation pipeline composed of three compute units. This paradigm forgoes the bulky Batch layer of the Lambda architecture and transfers its accuracy to the Speed layer. Unlike the previous setup, the data entries here are immutable and the introduction of rectifying data items (in blue) is the only way to amend an erroneous input (in red). The data are handled in the same fashion as the Speed layer (in green) of the Lambda architecture .

engines are needed. Although not immediately obvious, that subsequently entails a double administrative policy, which again adds to the overheads of running the infrastructure.

4) KAPPA ARCHITECTURE

First introduced by Jay Kreps in 2014 [74], and then illustrated by Martin Kleppman at the 2014 Strangeloop conference [75], Kappa architecture is a processing model similar to traditional event-oriented programming. The Kappa architecture is a simplification of the Lambda architecture, in which the batch system is removed [17]. Kappa moves the core background processes of a database into the foreground: rather than storing the data itself, the system acts as a logging module, storing the ordered, immutable, series of operations that have transited through it. There are numerous implications of this new paradigm, compared with the Lambda architecture:

- A message logging system is required.
- No input data are lost, except if explicitly requested.
- Correcting an input is impossible, and instead a new record is created that will balance out the previous erroneous entry.
- Only a single version of each algorithm needs to be implemented.

This yields a fairly simple infrastructure (single time scale, no algorithm duplication) where the computation layer has to be efficient enough to maintain real-time performance, capable of keeping up with the continuous stream of data. Another theoretical point supposes the log/message system will be able to replay messages on demand. This is an important feature that enables performing re-computations needed to ensure the integrity of the database upon modification of an algorithm. This last aspect will not be extended here as the implementation is very much dependent on the underlying technology (see sub-section III-A3).

We have reviewed essential theoretical concepts that shape the landscape of technological solutions available for Big Data processing in real-time. Designers and end-users of such solutions must consider their individual constraints, and understand that these constraints dependent on: what is expected of the system as a whole, its expected robustness to failure, its ability to recover from fault and compensate for the loss of data, the intended throughput, etc. Applications of 'Big Data' and data streams have properties that will inevitably influence the decision to utilize one solution over another. We reviewed four compute paradigms, which form the backbone of most off-the-shelf solutions. These paradigms attempt to accommodate hardware and software constraints to maximise performance and availability, by increasing redundancy of data, and structuring processing pipelines in order to leverage low cost computing architectures. Adopting such solutions comes at a cost, which depends on the application requirements, i.e. the use of off-the-shelf technologies is more cost effective than for the application tailored specific solutions.

III. AVAILABLE SOLUTIONS

As the rise of Big Data yielded new challenges, the number of off-the-shelf solutions grew in similar proportions. In this section, we review a selection of technologies; although these might rapidly become obsolete, our objective is to use these solutions to qualitatively map the technologies currently available. This qualitative analysis is important as it highlights the aspects that service providers believed were important for Big Data, and have thus far survived market pressures. To begin, we differentiate two types of solution: (1) the specialized modules and (2) stacks considered 'off-the-shelf'. This first distinction is important as each module, when considered separately, only covers a small portion of the problems to be addressed when creating a fully fledged

computational, fault-tolerant system. We do not discuss the different implementation-specific limitations of the middleware as these bear little weight with respect to the paradigm and architecture they represent.

A. MODULAR SOLUTIONS

Amongst the modules available, we can differentiate a number of roles or aims - note these are not necessarily mutually exclusive.

1) COMPUTE ENGINES

Processing capabilities is one of the pillars of Big Data. Many types of compute engines exist (e.g. Apache Hadoop) with each implementing the MapReduce or ''real-time'' paradigms to a degree. Apache Hadoop is commonly thought to be the embodiment of the Lambda architecture, which itself is often mistaken for a simple batch processing architecture, disregarding the benefit of the speed layer. Apache Hadoop laid the foundation for many different compute engines and merits a detailed discussion.

Apache Hadoop (see Table 1) is a compute cluster model and implementation based upon the MapReduce paradigm [76]. Its main role is the delivery of the (to be processed) data blocks to the computational nodes available and the management of the activity of these nodes. This is achieved using the network-distance sensitive Hadoop Distributed File System (dubbed HDFS) [52]. HDFS offers file splitting and redundancy across the network, along with local rack-awareness, enabling a smart triple-point redundancy of the data: one copy on the node to perform the computation, another on a neighbour node located in the local rack, and a third located in a foreign rack to shield against power outage (note that while other file systems are supported (see Table 1), some of the redundancy functionality might not be operative for other file systems than HDFS). Apache Hadoop architecture layers are as follows:

> **Bi-layered**: the HDFS layer carrying the data are distinct from the MapReduce layer handling the computational tasks
>
> **HDFS layer**: contains two types of nodes, the NameNode (controls scheduling and the indexation) and the DataNodes (act as containers, and are the entities through which data persistence and redundancy is implemented).
>
> **MapReduce layer**: also contains two types of nodes, the JobTracker (the gateway to job submissions and recipient of the rack-awareness information), and the TaskTracker (manages the health and location of a task). The TaskTracker node spawns a separate Java Virtual Machine (JVM), efficiently shielding the rest of the system from any operational crash. Any forcefully terminated task (e.g. crash, fail, time out) is rescheduled, leading to an at-least-once processing guarantee.

In its prime mode, Apache Hadoop presents itself as a FIFO queue, treating the tasks in order of arrival. This can be extended by specifying one of five optional scheduling priorities to favour some tasks over others; It is also possible to define a customised scheduler that is better suited to a particular workload. Additionally, its file system is designed to hold very large amounts of data (tens of petabytes) - while being possibly limited by the RAM requirement of the namespace [77]) making it a valid data storage solution for Big Data applications.

Apache Spark (see Table 1) [78] is considered the state of the art of the MapReduce paradigm. It was developed in response to one of the main limitations of the paradigm: the heavy reliance on I/O operations to/from the disk to perform batch computation. Apache Hadoop compute clusters are inherently hard drive intensive because of the I/O oriented redundancy and data: the *data loading* $\rightarrow$ *map computing* $\rightarrow$ *result storage* life cycle of the basic MapReduce model becomes a time consuming process, especially for tasks requiring successive iterations over the same dataset, i.e. Machine Learning model training. Storing and keeping immutable data in RAM directly reduces the overhead of the model. Note that while machine learning training is an iterative task, and could therefore be represented as a cyclic computational graph, as for most stream engines presented here, Apache Spark only supports acyclic computational graphs to be processed (termed DAG, Directed Acyclic Graph).

Apache Spark augments both the MapReduce paradigm and Apache Hadoop through the use of distributed data objects [78], which provide degrees of fault tolerance and can accommodate a range of querying languages through APIs.

All of the engines described so far are both particularly well-suited and limited to batch processing.

Apache Spark Streaming (see Table 1) [79] moves further towards real-time computing by enabling Apache Spark to support micro-batch processing. It is set up by periodically splitting any continuously incoming flow of data, and them to the Spark nodes. This naturally decreases the latency delivered by the engine as the maximum idle waiting time for a newly submitted dataset reduces hours to seconds.

The recent demand for real-time analytics and applications has driven a paradigm shift to addressing messages instantly upon submission and/or in order of production. This is one more step beyond Apache Spark Streaming, in that the concept of a batch is substituted by that of a single data record - as mentioned earlier, such an approach has a number of practical difficulties but constitutes the closest implementation of a resilient computation engine that bears no latency.

Apache Storm (see Table 2) [80] is a distributed-stream processing engine encoding the operations to be carried out as an acyclic graph through which records (termed events) - enter, flow and leave 'transformed'. Unlike Apache Spark, an event is not guaranteed to be treated at-least-once nor exactly-once. This restriction can be bypassed with **Apache Storm Trident** that provides an exactly-once processing guarantee, by replacing the event-driven computation paradigm with a micro-batch treatment. Apache Storm keeps within memory the 'history' of the graph operations that

TABLE 1. A comparison of the qualitative features of the Apache Hadoop, Apache Spark and Apache Spark Streaming compute engines. ***Programming Level*** defines the level of hardware abstraction; ***Native file system*** file systems for which the engine was originally designed; ***Alternative file system*** file systems that are now compatible, using with plugins to extend the original compatibility; ***Fault tolerance*** the presence/type of computational preservation mechanism; ***Fault tolerance type*** the fault tolerance qualification; ***Semantic*** the level of computational guarantee the engine delivers; ***Compute type*** the compute engine method for handling tasks as they arrive.

	Apache Hadoop	Apache Spark	Apache Spark Streaming
Programming Level	Low	Mid	Mid
Native file system	HDFS	HDFS, HBase	HDFS
Alternative file systems	FTP, Amazon S3, WASB	Shared file system	FTP, Amazon S3, WASB
Fault tolerance	Yes	Yes	Yes
Fault tolerance type	Re-scheduling	Re-compute operation line	Re-compute operation line
Load balancing	Yes	Yes	Yes
Semantic			EO
Compute type	Batch	Batch	Stream through microbatch
Operating systems	Windows, Linux	Windows, Unix-based	Windows, Unix-based
Native language	Java	Java, Scala	Java
Alternative language	Any (wrapper)	Python, R, Julia	Scala, Python, Clojure, R
Ambivalent database/Data model	HDFS	SQL, Dataframe, Cassandra, HBase, Hive, Tachyon, files (CSV, JSON, Parquet, ORC, Avro, text), any hadoop datasource	SQL, Dataframe, Cassandra, HBase, Hive, Tachyon, files (CSV, JSON, Parquet, ORC, Avro, text), NFS, GSC, any hadoop datasource
Source database and data providers	None	None	Amazon Kinesis Streams, Twitter, TCP sockets
Sink database and data providers	None	None	
Processing order	FIFO	FIFO	FIFO
Expected throughput	Low	Mid	Mid
Latency (unit of time)	Hours	Minutes	Seconds
License	Apache 2.0	Apache 2.0	Apache 2.0

TABLE 2. A comparison of the qualitative features of computer engines: Apache Storm, Apache Flink and Apache Samza. ***Programming Level*** defines the level of hardware abstraction; ***Native file system*** file systems for which the engine was originally designed; ***Alternative file system*** file systems that are now compatible, using with plugins to extend the original compatibility; ***Fault tolerance*** the presence/type of computational preservation mechanism; ***Fault tolerance type*** the fault tolerance qualification; ***Semantic*** the level of computational guarantee the engine delivers; ***Compute type*** the compute engine method for handling tasks as they arrive.

	Apache Storm	Apache Flink	Apache Samza
Programming Level	High	High	High
Native file system	Distributed FS	HDFS, S3, GCS, MAPRFS (all through FS abstraction)	Any
Alternative file systems	None	Any distributed FS	None
Fault tolerance	Yes	Yes	Yes (using Apache Yarn/Kafka)
Fault tolerance type	Re-compute from earliest failure point onward	Re-compute from latest failure point onward	Snapshot + Rescheduling
Load balancing	Yes	Yes	Yes (using Apache Yarn or Kafka)
Semantic	AMO, ALO, EO (through Trident)	EO	ALO
Compute type	Stream	Stream	Stream
Operating systems	Windows, Unix-based	Windows, Unix-based	Windows, Mac OS X (dev) and Linux (dev + prod)
Native language	Clojure, Java	Java, Scala	Java, Scala
Alternative language	Ruby, Python, Javascript, Perl, any language using Apache Thrift	Python	JVM languages
Ambivalent database, Data model	SQL	Apache Kafka, RabbitMQ, Amazon Kinesis Streams, Apache NiFi	Apache Kafka
Source database and data providers	Any through spout abstraction (a large number exist).	Twitter	Any (through plugable API)
Sink database and data providers		Elasticsearch, HDFS, Apache Cassandra, Redis, Flume, ActiveMQ (via Apache Bahir)	
Processing order	Time based	Time based	Time based
Expected throughput	High	High	High
Latency (unit of time)	milliseconds	milliseconds	
License	Apache 2.0	Apache 2.0	Apache 2.0

generated the data, such that in the event of a job failure, this information can be used to restart the job from the earliest failure point, ensuring operation completeness.

Apache Flink (see Table 2) [81] is a relatively recent stream processor exposing an only-once semantic and supporting acyclic computational graphs. It is described as being dedicated to 'distributed, high-performing, always-available, and accurate data streaming applications' [82]. Unlike Apache Storm, Flink ensures ordered data processing based upon the timestamp of data production rather than the time of arrival. Another strength of Apache Flink is the way it handles backpressure (where downstream operators are not

fast enough to process data at the same speed as the upstream operator pushing the data):

In the case of two workers hosted on the same machine, this process is achieved in memory whereas over the network, the reading process unfolding over TCP connection is interrupted in the event of a full buffer, effectively stalling the source until more capacity is available.

As for previously mentioned engines, those operations are carried out in a fault-tolerant fashion. A system of stream-replay and distributed check-pointing [81] tracks the unfolding of each task at the local and global scale and flags the progression of the different operations involved. This enables the rollback and recovery of a task in cases where one element is faulty.

Apache Samza (see Table 2) [63] is another relatively recent stream processing engine. It relies on Apache Kafka for data communication (see Section III-A3) and Apache Hadoop Yarn for fault tolerance, processor isolation, security and resource management (see Section III-A2). As such it is more of a stack than an independent compute engine, and this was its intended purpose when LinkedIn developed it concurrently with Apache Kafka.

Apache Samza resembles Apache Flink, but a key functional difference is its inability to perform batch processing: the engine is built entirely on the concept of streams and ordinary queue distribution. Another difference is its reliance to the stream partition functionality of Apache Kafka, to achieve cross-machine task parallelism; the tasks themselves are isolated within a process of a Linux cgroup analogously to Apache Storm. Finally, Apache Samza cannot guarantee an exactly-once semantic - although each element is ensured to be treated, the fault tolerance and distribution model only provide an at-least-once semantic.

2) COORDINATORS

We define 'coordinators' as an umbrella term to describe solutions in charge of maintaining the computation cluster and/or task execution integrity. For instance, the software might be used for another middleware to safely maintain their configuration parameters, or for a task scheduler to maintain the execution integrity of a group of computational instances. Nonetheless, they form the central coordination piece of the cluster, without which the good behaviour of the system is impossible.

Apache Hadoop YARN (see Table 3) [83], or 'Yet Another Resource Negotiator' [84], is a task scheduler coupled with a resource dispatcher for distributed processing systems. It is composed of two main components:

> **Resource Manager**: the master authority of resource and task distribution in the system.
> **Node Manager**: a per-machine task and resource monitoring agent in charge of providing useful metrics to the Resource Manager.

Within the Resource Manager agent there are two services in constant communication: the Scheduler, responsible for allocating resources to the various running applications (with respect to constraints and queues) and the Application Manager, which handles job submissions (from their acknowledgement and container negotiation, to the management of fault tolerance). In the event of a failure forcing the restart of the system, two options enable the (partial) preservation of the service state, essentially making the event invisible to the user:

> **Non-work-preserving Resource Manager restart**, preserves the state of the service in an external database (Zookeeper, HDFS, LevelDB) and kills the existing jobs. Upon failure, the state is restored and the previously running application is re-started.
> **Work-preserving Resource Manager restart**, preserves the state of the service in an external database but does not kill the existing jobs. Upon restart, the state of the newly activated Scheduler is rebuilt using information provided by Node Managers and Application Masters.

Together, those communicating components form an adaptable, customizable environment in which tasks are maintained and can thrive. Despite this adaptability, the infrastructure can be difficult to manage, especially for users new to the system.

Apache Twill (see Table 3) [85], formally Apache Weave, is an abstraction layer to Apache Hadoop YARN that allows the use of its distributed capabilities through a programming model similar to threading. However, because of their heavy reliance on the Resource Manager, both Apache Hadoop YARN and Apache Twill can potentially suffer from a single point of failure.

Apache Zookeeper (see Table 3) [86] takes a different approach, presenting a distributed coordination service - initially a sub-project stemming from Apache Hadoop, it is now a first-level project. Zookeeper aims to offer a low-latency, high availability, fault-tolerant distributed computational network, devoid of single points of failure. Zookeeper attributes a name and a path to each node, much like any file system; the difference lies in the low storage size available ($\leq$ 10KB), which is dedicated to the persistence of the cluster configuration and identification of each node. As this is not a Big Data passing/replicating system, it is necessary to run this architecture alongside a database with which to share the actual data to process.

Furthermore, the communication architecture demands that at least $(N+1)/2$ nodes (where N is the total number of nodes in the system) are running at any time for the system to maintain its operation. This translates into a three-node Kafka cluster (running in Zookeeper) being able to bear a maximum of one node failure, alike a four nodes system.

The official FAQ recommends aiming for a five node system as the number of nodes has a negative effect on the data writing speed of the ensemble (and a marginally positive impact on the reading speed).

Apache Curator (see Table 3) Apache Curator is an Java/JVM client library for Apache Zookeeper that aims to ease the use of the latter through a high-level API framework. It is not a coordinator in its own right but packages a number of pre-implemented procedures fit to tailor the coordination of a cluster to specific needs [87].

HTCondor is a coordinator that offers additional capabilities, that merit mentioning it here. HTCondor can harness the computational capabilities of machines that are idle (such as unused desktop machines), combining them into a bespoke Beowulf cluster. It can dynamically detect available machines and elastically adapt the distribution of the computational tasks and related data to the size of the cluster [88].

3) MESSAGE PASSING SYSTEMS

Message passing systems are an important part of all Big Data systems: when the quantity of data to be managed increases such that a single system can not possibly handle it, their displacement and direction towards the recipient nodes can become a problem. Message passing systems handle the heavy task of fault-tolerant message delivery and must be horizontally scalable and well-behaved regardless of the amount of data.

Apache Flume (see Table 4) [89] is a streaming platform that aims to deliver information to the HDSF of Apache Hadoop, thus it uses a simple structure optimized for that task, based upon data streaming. Pictured as an information funnel, driving data from various sources towards a centralized sink, Apache Flume can handle any data source, including social media, telecommunication networks, email, databases, other streaming platforms or files. The fault tolerance within Apache Flume is delivered through two steps: (1) every incoming message is stored in a passive queue hosted by the HDFS; (2) when two or more Flume processes need to communicate, the message within the sender's queue is only deleted when the receiver has acknowledged receipt and storage of the message content.

Apache Kafka (see Table 4) [90] was originally developed by LinkedIn alongside Apache Samza and was designed to compensate for the lack of ubiquity of Apache Flume. It is a streaming platform enabling the easy interconnection of multi-type data sources, compute engines and data sinks.It is seen as the correct implementation of the Kappa architecture in that it generates streams of data from sources that are, in principle, never to be deleted. This enables any process to be replayed from the beginning if any processing needed to be amended.

Its foundations rely on Apache Zookeeper to coordinate its different stream brokers; the clients are directly notified by the coordinator of the availability of each data broker. This means that when facing a running cluster, as long as the client accounts for the list of brokers provided and tries them sequentially, it is guaranteed to successfully send a message regardless of the main receiver node's availability. Each message/event transmitted is then redundantly stored within a distributed fragmented log, ensuring fault tolerance. In that respect, Apache Kafka is an efficient storage system.

The streams produced are organised in (user defined) topics and handled by Producers (applications producing data) and Consumers (applications receiving data) paired through the simplest form of communication, and their API which enables writing and reading from a data stream, respectively. Extending those basic functionalities, results in a processing and a connection framework:

> **Kafka Stream API**, supports and promotes stateful stream transformations. It is similar to Apache Storm in the sense that this is achieved through the generation of an acyclic computation graph distributed within the cluster.
>
> **Kafka Connect**, enables the external communication to and from Kafka. It facilitates the ingestion of large amounts of data (such as entire databases), in addition to data assimilation on-demand; it can be run as a standalone process or as a scalable, fault-tolerant cluster service.

The advertised capabilities of Apache Kafka recommend the clients to use the Kafka API, enabling managing of error messages and buffering necessary to achieve robustness within the platform.

RabbitMQ (see Table 4) was one of the first mature (good level of features, client libraries, developer tools, documentation) message brokers on the market. Focusing on message transmission, over archiving/retrieving events, it is unlike Apache Kafka as it comes with a large variety of messaging protocols such as AMQP, originally developed to support STOMP, MQTT and HTTP. Unlike Apache Kafka, RabbitMQ does not rely on an external coordinator to manage its data brokers and consumers. Presenting a more monolithic face, it assumes a server-centric model where the clients are only expected to subscribe to a queue to start consuming/publishing messages. The server side (and therefore queue side) assumes the buffering and distribution of the messages to the appropriated consumers [91].

4) DATABASES

Databases are inevitably an important component of production-grade 'Big Data' environments. While not mandatory (data could simply be streamed and the results consumed by the end-user without retention), they are ubiquitous and constitute main data sinks/sources. Loosely speaking, we can differentiate the following types of databases:

- Relational DBMS (e.g. Oracle, MySQL, Microsoft SQL Server, PostgreSQL, SQLite)
- Document databases, NoSQL (e.g. MongoDB, Amazon DynamoDB, Apache Cassandra, CouchDB, HBase, Couchbase)
- Graph databases (e.g. Neo4J, Titan, Giraph, InfiniteGraph, DGraph)

TABLE 3. A comparison of the qualitative features of the coordinators: Apache Hadoop YARN, Apache Twill, Apache Zookeeper and Apache Curator. ***Programming Level*** defines the level of hardware abstraction;***Fault tolerance*** the presence/type of computational preservation mechanism; ***Fault tolerance type*** the fault tolerance qualification.

	Apache Hadoop Yarn	Apache Twill	Apache Zookeeper	Apache Curator
Type	Cluster Coordinator	Cluster Coordinator	Cluster Coordinator	Cluster Coordinator
Programming Level	Low	High	Low	High
Fault tolerance	Yes	Yes	Yes	Yes (through Zookeeper)
Fault tolerance type	State save/reconstruction	State save/reconstruction	Master election / reroll	Retry + Zookeeper
Load balancing	Yes	Yes	Limited	Yes
Operating system	(dev) Windows, Mac OS X, (dev + prod) Linux	(dev) Windows, Mac OS X, (dev + prod) Linux	(dev) Windows, Mac OS X, (dev + prod) Linux	(dev) Windows, Mac OS X, (dev + prod) Linux
Native language	Java	Java	Java, conf files	Java, configuration files
Alternative language	None	None	None	None
License	Apache 2.0	Apache 2.0	Apache 2.0	Apache 2.0

TABLE 4. A commparison of the qualitative features the streaming platforms: Apache Kafka, Apache Flume, and RabbitMQ. ***Programming Level*** defines the level of hardware abstraction; ***Native file system*** file systems for which the engine was originally designed; ***Alternative file system*** file systems that are now compatible, using with plugins to extend the original compatibility; ***Fault tolerance*** the presence/type of computational preservation mechanism; ***Fault tolerance type*** the fault tolerance qualification; ***Semantic*** the level of computational guarantee the engine delivers; ***Compute type*** the compute engine method for handling tasks as they arrive.

	Apache Kafka	Apache Flume	RabbitMQ
Type	Streaming Platform, Compute Engine	Streaming Platform	Streaming Platform
Programming Level	High	High	High
Native file system	N/A	HDFS	N/A
Fault tolerance	Yes	Yes	Yes
Fault tolerance type	Re-routing, data replication	data replication	data replication
Load balancing	Yes	Yes, at the sink level	Yes (Queues), No (nodes)
Semantic	Transaction: ALO (EO in dev.) Stream API: ALO, EO	ALO (EO in discussion)	AMO, ALO (conf. dependant)
Compute type	Stream	N/A	N/A
Operating system	Windows, Unix-based	Ubuntu, CentOS, Red Hat, Suse, Mac OS X	Debian-based, SUSE, Mac OS X, RPM-based, Windows XP and later
Native language	Java, Scala	Java, Scala	Erlang
Alternative languages (for clients)	C, C++, Python, Erlang, Go (aka golang), .NET, Clojure, Ruby, Node.js, Proxy (HTTP, REST), Perl, PHP, Rust, Alternative Java, stdin/stdout, Storm, Scala DSL, Clojure	Assumes clients can produce a Flume compatible message	Java, .net, Objective-C, Ruby, Python, PHP, Swift, Clojure, JRuby, JavaScript, C, C++, Go, Erlang, Haskell, OCaml, Perl, Common Lisp, COBOL
Ambivalent database/Data model	Virtually any	HDFS	Virtually any
Source database and data providers	N/A	Any	N/A
Sink database and data providers	N/A	None	N/A
Processing order	Time-based	FIFO	FIFO
Expected throughput	High	Mid	Mid
Latency (units of time)	milliseconds		
Open Source	Yes	Yes	Yes
License	Apache 2.0	Apache 2.0	Mozilla Public 1.1

- Multi-model databases formed from hybrids of the above (e.g. MarkLogic, Microsoft Azure Cosmos DB, OrientDB, Apache Drill, ArangoDB)

Regardless of the type used, the setup needs to be scalable and robust enough (according to the Volume of the data) to handle the high throughput of the computational system. As such, distributed implementation seems optimal, despite the constraints posed by the CAP Theorem (see Subsection II-C1).

The choice of database systems and their specifications is a vast topic and is outside the scope of this review. Multiple benchmarks and reviews [92], [93] highlight the capacity, strengths and weaknesses of existing databases, and the different documentations for the products cited above.

B. FULL STACK AND CLOUD COMPUTING

1) MANAGED PLATFORMS

The main conclusion that can be drawn from this review is that 'Big Data' and real-time analytics is a highly complex and interdisciplinary field that requires diverse expertise in network communication, IT infrastructure, storage, control and optimisation; this expertise is required even before one can begin to plan the types of processing and analytical pipelines that could yield return on investment from the data available. This expertise is likely to involve diverse teams, whose roles will become critical for core operations of businesses.

For companies that are not involved in research and development of computing solutions, this required expansion can have substantial cost, and several service providers

are offering simplified solutions, involving decentralised and remote resources that are managed on behalf of customers. The goal is to provide reliable and flexible solutions that can be adapted to needs as they arise.

This is essentially a paradigm shift, where one purchases the amount of resource needed for a particular task, as and when it is needed. This is in contrast to the more traditional model that sees hefty investments in hardware, software and human resources, in anticipation of future needs, that may or may not transpire. Dedicated computing hardware hosted in-house, comprising desktops and computing clusters, can be replaced entirely with cloud computing facilities that are virtually as powerful and versatile as needed. This resource can scale rapidly, on-demand, and provides extended capacity and cost benefits. This is a new and volatile market, however, and although the direction of travel is clear, the precise shape of the technological landscape is likely to see numerous changes in years to come.

This emerging market has been enabled by a range of technologies, and several examples are worth outlining here. Virtualization and containers, for instance, which abstract software requirements from the hardware layer, have opened up new possibilities. A particular set of software packages can be developed and maintained in a given environment, which can be reproduced and run elsewhere; the details of the hardware being used becoming largely irrelevant for most operations. Docker and Singularity are amongst popular choices. These solutions play a critical role in the landscape of technologies, and must provide resolutions to a wide range of issues.

Security, for instance, is a concern that has slowed down acceptance of these middleware solutions: as middle layers must negotiate access to privileged resources on unknown hardware, which in most cases is shared amongst groups of users, permissive processes can become attack vectors for malicious actors. Another such issue concerns the replication of a given environment which may need low-level reconfiguration, involving compilation steps that can introduce small discrepancies in code and during execution. Such issues can be deal-breakers for specialist applications that cannot afford uncertainty. Cloud computing platforms, however, have received a lot of attention, and keep growing rapidly. It is predicted that the global cloud computing market will value over $200B in 2019, growing over 50% from 2018 [94].

Abstraction layers afford a range of new opportunities to developers, and come in various forms, ranging from bare-metal cloud-server solutions effectively providing a basic operating system on shared hardware (i.e. Rackspace), that can be reconfigured on-the-go with a user interface (Amazon Web Services (AWS), Microsoft Azure, Nutanix), up to modular access to specific tools that manage and optimise network communication, data storage or even out-of-the-box analytical pipelines (i.e. Google Cloud Platform). These tools may monitor communication between decentralised entities, rearrange and move data in response to resource outage at a particular geographical location, organise communication queues to minimise packet loss, or adjust the processing pipeline. Developers thus trade convenience for flexibility, leaving the system to manage itself. A consequence of this evolving situation is that users tend to become 'vendor-locked', due to the cost of redeveloping code that does not rely on these convenience mechanisms.

2) TASK SCHEDULING AND RESOURCE ALLOCATION IN THE CLOUD

In the paradigm where a company budgets and purchases only as much resource they need, for exactly the time they need it, it falls on the provider to ensure that task scheduling and resource allocation is organised and managed efficiently, and at a reasonable cost. This is particularly important for real-time analytics of data streams because the characteristics of the input flows are likely to vary over time (see Section II-A), and providers are thus required to be responsive and flexible to demand whilst keeping costs low [95], [96]. Our review takes the perspective of the user, not that of the provider. Thus describing in detail the paradigms deployed by the providers falls outside the remit of the work presented here. We feel it is valuable, however, to become familiar with the landscape of approaches that providers may deploy, and the focus of what they see as important. This is not, however, an easy task: big companies, like Google [97], Amazon, Microsoft, or even Facebook [98] or Twitter, tend to keep most of the details of their technology under embargo for significant periods of time, for evident reasons. This technology, which may be available for hire, may also be subject to change at a moment's notice.

For cloud solutions providers, decisions supporting the provision of computing resources only make sense in the context of their own business plans, and the way they may drive profits. Margins typically stem from a range of sources, spanning customer-facing features (like an AI-based API, or a robust lossless network queue) and back-end optimisations (like the efficient management of virtualized resources, or geo-localised redundancies of data centers). In addition, although the democratization of AI and data are now pervasive in virtually every sector, for any given user, the development of a cloud-based product remains a significant endeavour, and thus providers have the hard task of streamlining their offering, ease access and remove hurdles to adoption.

On the one hand, sources of margins are tangible assets, often available to customers for comparison. On the other hand, costs minimization strategies are more opaque and complex, and ensuing technological decisions may bear significant weight on scheduling capabilities for any given cloud provider. In addition to the more typical costs of running a global company, cloud providers must manage the costs involved of running their infrastructure, providing users with security, robustness and redundancy. Costs associated with energy consumption, for instance, and associated task scheduling decisions can make or break a provider [99], [100]. It is also worth noting that applications that are now

common, like your typical natural language processing (NLP) feature, used by personal assistant systems on mobile devices, bear significant carbon footprint. It was for instance demonstrated that the training alone of a common NLP deep learning network can be equivalent to the carbon emissions of five cars in their lifetimes [101]. Such studies provide ground for the development of so-called Green Cloud Data Centers, which attempt to maximise revenue for cloud providers whilst at the same time minimizing energy cost and carbon footprints [102].

Global cloud providers do attempt to design solutions that provide flexibility in dealing with such issues. At the time of writing, a good example of ''out-of-the-box'' thinking is exemplified by the Dremel architecture that powers several core services at Google [103], [104]. Unlike typical tabular data that is accessed sequentially, the Dremel architecture is characterised by two features that are designed to reduce response time for big data, and are optimised for MapReduce-based computing architectures and applications.

- Columnar storage: Data records are split and stored in columns, not rows as has been customary in databases. This innovation has the advantages that the high degree of redundancy in any given column can be compressed, and that queries can thus manipulate small subsets of the whole data. A disadvantage, however, is that the architecture is optimised for retrieval and updates can thus be extremely costly.
- Processing tree architecture: In addition to a high degree of storage redundancy, the processing of a query relies on decentralised sub-queries that are operated in separate nodes (leaves). Sub-queries are then simply consolidated in a tabular way, if and when needed.

It is our belief that years to come will see the industry move in that direction, proposing technologies in response to common customers' needs, such as lossless network queues, as well as entirely new ways to structure data to align with current on-demand paradigm for flexible applications.

IV. PRACTICAL CONSIDERATIONS

A. DECISION POINTS

In light of the difficulty comparing platforms on an equal footing, and short of realistic and meaningful benchmarking, one relies on theoretical considerations to inform decisions on architecture to implement. We can imagine that, for every problem, it is possible to formulate a Kappa or Lambda architecture oriented solution based on the fact that (1) the Kappa architecture is essentially a refined and persistent version of the Lambda's speed layer, tasked with compensating for the latency of the batch engine and (2) that the Lambda architecture can be applied to any problem, even in extremely inefficient fashion. We, therefore, provide here a set of aspects that indicate if a particular problem is more efficiently solved using one architecture or the other.

Let us begin with **latency**: if the application to be designed holds low latency constraints, the simplest approach is the Kappa architecture which treats the elements as they arrive with minimal delay.

Second, we move to the data **time dependency**: if the application needs to integrate over an infinite period of time to formulate an answer, then Kappa architecture will present the simplest implementation, through the use of exponential decay (i.e. Reinforcement Learning) of incremental assimilation. Alternatively, if the time dependency is limited, the Lambda architecture is a better choice: the batching of the data reduces the overhead associated with a single, isolated, computation instantiation - it requires set up once per batch, instead of once per event.

Third, the **complexity of the algorithm** is taken into account. The Lambda architecture requires a double implementation of each algorithm to compensate for batch layer latency, which means that the more complex the algorithm, the harder and more expensive it is to maintain. Subsequently, the need for **hardware maintenance** is evaluated: most compute engines capable of handling the Kappa architecture can only tolerate a certain number of failed nodes before experiencing service discontinuity. The Lambda architecture features a natural inactivity period (node dependent - between batches), that is minimised by the scheduler (see subsection II-B), where most of the compute nodes are at rest making it easier to schedule a mass maintenance operation.

Finally, we consider the **recursive nature** of the algorithm: in any architectural case, this aspect can induce problems. Some compute engines (Apache Spark, Apache Spark Streaming, Apache Flink) come with features enabling the iteration over the same set of data without the overhead of loading the data from the disk. However, this feature is not part of the architectural design of these paradigms as it constitutes an application specific case. In the absence of such a feature, the Message Passing System can be employed as remedy, continuously rewriting the same piece of data within the streaming pipeline or shifting the reading index backward accordingly. Unfortunately, this is just a patch and more research is needed to develop a more durable solution.

B. REAL-LIFE ILLUSTRATIONS

As a result, multiple integrated hosting platforms (e.g. Microsoft, Amazon, Google) offer compute paradigms on demand, enabling them to cover as many applications as possible. Those architectures, when combined, permit the elaboration or addition of numerous technologies. In an over-simplified view, our computerised financial world works in such a way: the Real Time Gross Settlement System (RTGS) is a real-time system handling important transactions as soon as they are submitted (gross settlement). These are usually large economy critical systems operated by the central bank of a country, but on the other hand, payment systems in charge of handling smaller transactions batch them for periodical clearing. Similarly, telecommunications hot-billing exploits near real-time systems to track service usage by individual users of prepaid packages whereas billing for postpay packages can be batch assessed after a fixed period.

As the maturity of the available middlewares and the computational power increases, more applications will be within the grasp of businesses. The Internet of Things (IoT), which refers the widening network of ordinary objects communicating, comes with the production of a massive amount of heterogeneous information that can be relevant to a multitude of fields. For instance, while most IoT devices focus on usability and interactivity, the large-scale aggregation of basic information, such as whether or not the device is running, could open the door to pattern identification, consumption extrapolation and eventually, to more efficient power grid management systems. This would enable the ingestion of high volume of data produced by smart homes in nearly real-time.

C. DATA STREAM MINING TECHNIQUES AND ALGORITHMS AND MOBILE DATA MINING

For over a decade, research has been conducted on Data Stream Mining (DSM) algorithms to address the 'Velocity' aspect of Big Data. The two main drivers behind these developments are: (1) rapid model construction and the update of techniques to cope with high velocity data streams; (2) changes in the pattern encoded in the stream (also known as concept drift) which data mining algorithms aim to learn. The advancement of DSM algorithms has enabled the implementation of Data Mining applications on Mobile Devices, smartphones and tablets, leading to distributed mobile data mining systems.

1) ADAPTING BATCH DATA MINING TECHNIQUES TO ANALYSE DATA STREAMS

Several approaches have been proposed to modify existing data mining techniques for their application on streaming data, such as windowing, sliding windows [105], [106] and reservoir sampling [107]. The underlying concept of windowing or the sliding window approach is to only consider recent data instances for building data mining models. A variable size sliding window technique is the ADaptive sliding WINdow method (ADWIN) [108]. ADWIN changes the window size based on observed changes: if there are changes then the window shrinks, if there are no changes it increases. A drawback of ADWIN is that windows can potentially become very large which results in longer time requirements for adaptation. Reservoir sampling maintains and updates a representative sample of all the data in stream, and can be used to build data mining models representative of all the data observed so far. However, models built using reservoir sampling assume that each data instance, regardless of when it was generated, is equally represented in the pattern encoded in the stream.

Standalone concept drift detection methods exist, and are typically used in combination with batch data mining algorithms and sliding window approaches. For example, the sequential analysis technique CUmulative SUM [109] detects a drift when the mean of incoming data deviates significantly. The Exponentially Weighted Moving Average (EWMA) uses charts to monitor the mean of misclassification rates of a classifier, and gives less weight to older data instances and greater weight to more recent data instances. A traditional concept drift detector, named Drift Detection Method (DDM) [110] computes error statistics over two consecutive time windows, however, this method tends only to detect sudden drifts while gradual drifts are frequently missed [111]. The Early Drift Detection Method (EDDM) [112] is a further development of DDM; its drift detection is based on estimating the distribution of distances between classification errors, however EDDM is very sensitive to noise. A more recent concept drift detection method tailored for machine learning uses statistics about the extent to which models are modified by newly arriving data [113].

2) ADAPTIVE DATA STREAM MINING ALGORITHMS

Most of these algorithms can be broadly categorised into classification and clustering techniques. This section discusses Data Stream Mining algorithms that are adaptive to concept drift without the need for a separate concept drift detection or windowing method. Other more specialised algorithms exist, but are outside the scope of this review.

a: DATA STREAM CLASSIFICATION TECHNIQUES

A notable family of data stream classification techniques are the *Hoeffding bound* based techniques. The Hoeffding inequality provides an upper bound for the probability of the sum of a random variable diverging from an expected value. This Hoeffding bound has been used successfully for the development of various data stream mining algorithms known as Very Fast Machine Learning (VFML) techniques [114]. An additional development of classification techniques based on the Hoeffding bound is the Hoeffding Tree family of algorithms: the original Hoeffding Tree algorithm was able to learn incrementally in real-time [115] and has since been improved in terms of accuracy and data processing speed into the Very Fast Decision Tree (VFDT) algorithm [116], which can adapt to concept drift, by further expanding the tree. Unfortunately, this adaptation is very limited as previously learned concepts are not forgotten, thus the CVFDT algorithm (where C is for Concept Drift) was developed and removes this limitation by using a sliding window approach to alter entire subtrees. It is important to note that the tree structure makes tree-based data stream classifiers susceptible to noise. Alternatively there are rule-based data stream classifiers, such as VFDR [117] or Hoeffding Rules [118], which have more modular classification models. They may not reach the same levels of predictive accuracy as tree-based approaches but are generally more robust to noise. Additional approaches include APSO based on particle swarm optimization [119], Prototype-based Classification Model [120], SFNClassifier [121] and ensemble-based frameworks [122]. There are constantly more methods being developed and there is no single classifier that works well on all data stream sources. Therefore it is important for the analyst to have a good tool-set of algorithms available to tailor predictive data stream mining workflows to specific application needs.

b: DATA STREAM CLUSTERING TECHNIQUES

Cluster analysis is the method of grouping of data with similar characteristics, with the aim of having a high degree of similarity within a cluster, but a high degree of dissimilarity between clusters. With respect to cluster analysis on data streams, clustering algorithms must have a short processing time, ideally one pass through the data to adapt to concept drifts should be sufficient; and they also need a minimum memory footprint (as not all observed instances can be kept in memory). To date, several such algorithms have been developed, and an early algorithm in this direction is BIRCH [123]. Rather than storing raw data, BIRCH generates, stores, and updates statistical summaries of clusters; in this manner, BIRCH can learn incrementally but cannot forget obsolete concepts. These statistical summaries are known as Micro-Clusters and many data stream cluster analysis algorithms are based on some form of Micro-Cluster data structure; typically they hold as many of these structures in memory as computationally viable, and on demand, in an offline process, can build actual clusters by treating Micro-Clusters as data instances.

In this sense a notable development of BIRCH is the CluStream algorithm [124] which extends Micro-Clusters to include a time component, which enables CluStream to detect obsolete Micro-Clusters and thus forget obsolete concepts and browse through historical models. Another extension of BIRCH is the ClusTree algorithm, which implements hierarchical data stream clustering, however a major shortcoming of CluStream is that these clusters are circular, whereas many problems require alternative shapes. To address this, the DenStream algorithm [125] introduces 'dense' Micro-Clusters to summarise clusters using an arbitrary shape. In addition the authors of [126] developed a density based clustering algorithm in combination with rough sets. There is also a Micro-Cluster based development for the purpose of classification, the MC-NN algorithm, in which a new Micro-Clusters data structure has been developed to parallelise data stream classification in order to scale to multi source and high velocity data streams [127]. A recent data stream clustering approach is the A recent Micro-Clusters based development, for the purpose of classification, is the MC-NN algorithm, in which a new Micro-Clusters data structure has been developed to parallelise data stream classification in order to scale to multi source and high velocity data streams [127]. As for data stream classification many more data stream clustering algorithms exist, and it is important for the analyst to have a strong toolkit of algorithms available so they may tailor workflows to the specific needs of the application at hand.

3) DATA STREAM MINING ON THE GO IN EDGE ENVIRONMENTS

A discussion of streaming analytics in mobile environments is also necessary due to the growing importance of edge computing. Edge computing refers to distributed computing, in which data storage and processing is kept closer to the devices at which data are recorded, so-called edge devices [128]. The purpose of edge computing is to optimise computation by reducing data communication delays. In IoT applications such edge devices are often sensors, smartphones, tablets. The combination of edge computing with Data Stream Analytics has been subject in Data Stream Mining research for over a decade. Overall it is expected that with the roll out of G5 technology data driven machine learning applications will move closer into the edge then ever before [129]. Systems are realised either with specialised mobile hardware or off the shelf mobile devices such as Personal Digital Assistants (PDAs) or in recent years smartphones and tablets. There have never been better opportunities to leverage the increasing computational power of such devices, owing to their increasing computational and storage capacity, readily available sensor systems such as gyroscopes, cameras, microphones, and connectivity such as Wi-Fi, mobile internet, Bluetooth, etc. Formerly core functionalities such as phone calls and text messages, seem nowadays just like an additional feature rather than core functions. Such mobile data mining systems started off with basic Mobile Interfaces executing data analytics tasks server side but then quickly moved into on-board execution and hybrids [130].

An early representative of mobile data mining systems is *MobiMine* developed in 2002 [131]. MobiMine offered mobile on the go analytics capabilities for stock market prices. MobiMine is based on computationally limited PDAs which had little computational power at the time and thus presents a Mobile Interface and computational tasks are executed on a server architecture. The same group that developed MobiMine developed the *Vehicle Data Stream Mining System (VEDAS)* [132] using PDAs on board of moving vehicles to monitor driving behaviour using pattern extraction from streams to detect abnormal behaviour. Its commercial pendant is *MineFleet* [133]. As smartphones became available capable of recording and processing data in real-time through their sensor and processing capabilities, software systems that allow execution of algorithms on smartphones have emerged, like the *Open Mobile Miner (OMM)* [134]. Shortly after the *Pocket Data Mining System (PDM)* framework appeared as a first proof of concept that exploration and collaborative data mining is possible in streaming environments [130], [135]. Since PDM various smartphone based data stream mining technologies emerged, such as *Mobile Sensor Data Engine (MOSDEN)* [136]. MODSEN similar to PDM and attempted to create a mobile collaborative analytics platform optimised for sharing data across multiples applications and users to make use of smartphones' sensory capabilities. Another such development is *UniMiner* [137]. UniMiner aims to achieve scalability of data mining tasks through hybrid execution models, leveraging computational power of wearables, smartphones and the cloud, aiming at maximising data processing at the source, thus minimising data communication cost. The data ***Red****uction on the* ***Edge*** *architecture (RedEdge)* [138] also aims at computing data in the edge close to its source. RedEdge makes use of IoT devices as its primary data pro-

cessing environment. However, in case of unavailability, i.e. due to low battery, it offloads data processing to either near devices or the cloud.

D. CONJECTURES

Considering the surge in IoT currently occurring worldwide, an approach similar to telecommunication billing could be imagined in other left-behind fields, such a water and electricity billing. We still have, in 2020, to handout the reading of the different meters to the different provider for them to be able to approximate our periodic billing. This practice is, in our opinion, certainly a legacy of the pre-connectivity era that required human action for any data collection process, coupled with the heterogeneity of the entities managing the infrastructures.

On the other hand, while this usage suggestion follows the trend of the ever-growing data collection and IoT devices multiplication, we foresee potential network issues stemming from the exponential growth of those appliances. While it is customary to use low power communication interfaces for IoT devices, this only limits the amount of data that can be emitted on the LAN, per device. However, considering their number and coupled with the young age of the industrial market (no unified communication protocol standard, no enforced security policies [139], [140], limited understanding of the technology), we might not be able to anticipate the appearance of billions of ever-measuring new devices on the network, constantly connected to the cloud [141].

This will require ever more elastic infrastructure and scheduling to prevent any discontinuity of services: as data processing infrastructure become more and more vital for the diverse companies, optimizing the costs and maximizing the availability will prove ever more critical to the business. Current compute stacks and architectures already put the emphasis on those characteristics through the use of node redundancy and job re-scheduling. However, most of the distributed systems currently available - at least in the open source domain - are not natively elastic and require a custom implementation/stack to achieve this model.

V. CONCLUDING REMARKS

Big Data Technologies and Analytics for Data streams has recently emerged as a field of study, due to the development of IoT and similar applications. There is a vast landscape of technologies, platforms and applications for Big Data, however, they vary widely and can often build upon fundamentally different principles. This review aimed to organise the landscape of technologies, by applying some form of classification or categorisation, and discussed the advantages and limitations of different approaches. In this context, the review discussed computational concepts, compute paradigms, and some of the available solutions, and provided a pratical discussion of the strengths, weaknesses and limitations of these with respect to their functions, applications and what would constitute the 'ideal' system. The paper aimed to emphasis the computing principles of the technologies rather than the individual distributions. The reason for this is that the computing principles enjoy a much increased longevity than the individual distributions. Often already existing computing principles or paradigms can be found in new technologies. Another aspect discussed is the scalablity and flexibility of existing technologies with respect to the surge of IoT applications and devices. Here a lack of unified communication standards and heterogeneous security policies may overburden existing technologies in the near future. A more flexible infrastructure and easy to use/implement systems are desirable to ensure continuity of services and maximizing availability.

ACKNOWLEDGMENT

The authors would like to thank and acknowledge valuable feedback and comments from Dr. Orla Fannon.

REFERENCES

[1] B. Brown, M. Chui, and J. Manyika, "Are you ready for the era of 'big data,'" *McKinsey Quart.*, vol. 4, no. 1, pp. 24–35, 2011.

[2] S. Lohr. *Opinion | Big Data's Impact in the World*. The New York Times. Accessed: Feb. 2012. [Online]. Available: https://www.nytimes.com/2012/02/12/sunday-review/big-datas-impact-in-the-world.html

[3] Z. Lv, H. Song, P. Basanta-Val, A. Steed, and M. Jo, "Next-generation big data analytics: State of the art, challenges, and future research topics," *IEEE Trans. Ind. Informat.*, vol. 13, no. 4, pp. 1891–1899, Aug. 2017.

[4] F. Suchanek and G. Weikum, "Knowledge harvesting in the big-data era," in *Proc. Int. Conf. Manage. Data (SIGMOD)*. New York, NY, USA: ACM, 2013, pp. 933–938, doi: 10.1145/2463676.2463724.

[5] R. Pepper and J. Garrity, "The Internet of Everything: How the network unleashes the benefits of Big Data," in *Rewards and Risks of Big Data* (Global Information Technology Report), no. 13. Cologny, Switzerland: World Economic Forum, 2014, pp. 35–42. [Online]. Available: http://reports.weforum.org/global-information-technology-report-2015/

[6] C. L. P. Chen and C.-Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: A survey on big data," *Inf. Sci.*, vol. 275, pp. 314–347, Aug. 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020025514000346

[7] J. R. Mashey. *Big Data and the Next Wave of InfraStress Problems, Solutions, Opportunities*. Accessed: 1998. [Online]. Available: https://www.usenix.org/conference/1999-usenix-annual-technical-conference/big-data-and-next-wave-infrastress-problems

[8] V. Parmar and J. Yadav, "Big data: Meaning, challenges, opportunities, tools," *Int. J. Adv. Res. Comput. Sci.*, vol. 8, no. 1, pp. 165–168, 2017. [Online]. Available: http://www.ijarcs.info/index.php/Ijarcs/article/view/2875

[9] A. Gullberg, "Den samhälleliga självreflexionens möjligheter?: Big data på 1980-talet," in *Fragment 80-Tal*. Stockholm, Sweden: Nordiska Institutet för Samhällsplanering, 1991, pp. 180–187. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-149318

[10] P. M. Hartmann, M. Zaki, N. Feldmann, and A. Neely, "Big data for big business? A taxonomy of data-driven business models used by start-up firms," in *Proc. Cambridge Service Alliance*, Mar. 2014. [Online]. Available: https://cambridgeservicealliance.eng.cam.ac.uk/resources/Downloads/Monthly%20Papers/2014_March_DataDrivenBusinessModels.pdf

[11] D. D. Hirsch, "The glass house effect: Big data, the new oil, and the power of analogy," *Maine Law Rev.*, vol. 66, p. 373, Feb. 2014. [Online]. Available: https://papers.ssrn.com/abstract=2393792

[12] M. Kiran, P. Murphy, I. Monga, J. Dugan, and S. S. Baveja, "Lambda architecture for cost-effective batch and speed big data processing," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Oct. 2015, pp. 2785–2792.

[13] K. C. Sieben. *Labor Markets in 2040: Big Data Could Be a Big Deal for Jobseekers: Monthly Labor Review: U.S. Bureau of Labor Statistics*. Accessed: Feb. 2016. [Online]. Available: https://www.bls.gov/opub/mlr/2016/article/labor-markets-in-2040-big-data-could-be-a-big-deal-for-jobseekers.htm

[14] D. Alexander and K. Lyytinen, "Organizing successfully for big data to transform organizations," in *Proc. AMCIS*, Boston, MA, USA, Aug. 2017, pp. 1–10. [Online]. Available: http://aisel.aisnet.org/amcis2017/DataScience/Presentations/30

[15] L. Columbus. *IBM Predicts Demand For Data Scientists Will Soar 28% By 2020*. Accessed: May 2017. [Online]. Available: https://www.forbes.com/sites/louiscolumbus/2017/05/13/ibm-predicts-demand-for-data-scientists-will-soar-28-by-2020/#7a9e05e7e3bd

[16] M. Dave and R. Gianey, "Different clustering algorithms for big data analytics: A Review," in *Proc. 5th Int. Conf. Syst. Modeling Advancement Res. Trends*, Nov. 2016, pp. 328–333.

[17] W. Wingerath, F. Gessert, S. Friedrich, and N. Ritter, "Real-time stream processing for Big Data," *Inf. Technol.*, vol. 58, no. 4, pp. 186–194, 2016. [Online]. Available: https://www.degruyter.com/view/j/itit.2016.58.issue-4/itit-2016-0002/itit-2016-0002.xml

[18] A. Gattiker, F. H. Gebara, H. P. Hofstee, J. D. Hayes, and A. Hylick, "Big Data text-oriented benchmark creation for Hadoop," *IBM J. Res. Develop.*, vol. 57, nos. 3–4, pp. 10:1–10:6, May 2013.

[19] M. Chen, S. Mao, and Y. Liu, "Big data: A survey," *Mobile Netw. Appl.*, vol. 19, no. 2, pp. 171–209, Apr. 2014, doi: 10.1007/s11036-013-0489-0.

[20] Z. Zheng, P. Wang, J. Liu, and S. Sun, "Real-time big data processing framework: Challenges and solutions," *Appl. Math. Inf. Sci.*, vol. 9, no. 6, p. 3169, 2015.

[21] G. Li and X. Cheng, "Research status and scientific thinking of big data," *Bull. Chin. Acad. Sci.*, vol. 27, pp. 647–657, Jun. 2012.

[22] M. D. Assunção, R. N. Calheiros, S. Bianchi, M. A. S. Netto, and R. Buyya, "Big data computing and clouds: Trends and future directions," *J. Parallel Distrib. Comput.*, vols. 79–80, pp. 3–15, May 2015, doi: 10.1016/j.jpdc.2014.08.003.

[23] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. U. Khan, "The rise of 'big data' on cloud computing: Review and open research issues," *Inf. Syst.*, vol. 47, pp. 98–115, Jan. 2015, doi: 10.1016/j.is.2014.07.006.

[24] L. Columbus. *Big Data & Analytics Is The Most Wanted Expertise By 75% Of IoT Providers*. Accessed: Aug. 2017. [Online]. Available: https://www.forbes.com/sites/louiscolumbus/2017/08/21/big-data-analytics-is-the-most-wanted-expertise-by-75-of-iot-providers/#4daa1de51887

[25] J. L. Reyes-Ortiz, L. Oneto, and D. Anguita, "Big data analytics in the cloud: Spark on Hadoop vs MPI/OpenMP on beowulf," *Procedia Comput. Sci.*, vol. 53, pp. 121–130, Jan. 2015, doi: 10.1016/j.procs.2015.07.286.

[26] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "BigDataBench: A big data benchmark suite from Internet services," in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2014, pp. 488–499.

[27] M. Capota, T. Hegeman, A. Iosup, A. Prat-Pérez, O. Erling, and P. Boncz, "Graphalytics: A big data benchmark for graph-processing platforms," in *Proc. GRADES*. New York, NY, USA: ACM, 2015, pp. 7:1–7:6, doi: 10.1145/2764947.2764954.

[28] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis," in *Proc. IEEE 26th Int. Conf. Data Eng. Workshops (ICDEW)*, Mar. 2010, pp. 41–51.

[29] W. Xiong, Z. Yu, L. Eeckhout, Z. Bei, F. Zhang, and C. Xu, "Shen-Zhen transportation system (SZTS): A novel big data benchmark suite," *J. Supercomput.*, vol. 72, no. 11, pp. 4337–4364, Nov. 2016. [Online]. Available: https://link.springer.com/article/10.1007/s11227-016-1742-7

[30] D. Laney, "3D data management: Controlling data volume, velocity, and variety," META Group, Stamford, CT, USA, Tech. Rep., Feb. 2001. [Online]. Available: https://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf

[31] J. Gantz and D. Reinsel, "Extracting value from chaos," EMC Corp., Hopkinton, MA, USA, Tech. Rep., Jun. 2011. [Online]. Available: https://www.emc.com/collateral/analyst-reports/idc-extracting-value-from-chaos-ar.pdf

[32] *The Four V's of Big Data*. Accessed: 2013. [Online]. Available: http://www.ibmbigdatahub.com/infographic/four-vs-big-data

[33] B. Saha and D. Srivastava, "Data quality: The other face of big data," in *Proc. IEEE 30th Int. Conf. Data Eng.*, Mar. 2014, pp. 1294–1297.

[34] H. Hu, Y. Wen, T.-S. Chua, and X. Li, "Toward scalable systems for big data analytics: A technology tutorial," *IEEE Access*, vol. 2, pp. 652–687, 2014.

[35] I. Anagnostopoulos, S. Zeadally, and E. Exposito, "Handling big data: Research challenges and future directions," *J. Supercomput.*, vol. 72, no. 4, pp. 1494–1516, Apr. 2016. [Online]. Available: https://link.springer.com/article/10.1007/s11227-016-1677-z

[36] J. Gama and M. M. Gaber, Eds., *Learning From Data Streams: Processing Techniques in Sensor Networks*. Berlin, Germany: Springer-Verlag, 2007. [Online]. Available: http//www.springer.com/gb/book/9783540736783

[37] *Data, Data Everywhere*. The Economist. Accessed: Feb. 2010. [Online]. Available: https://www.economist.com/special-report/2010/02/25/data-data-everywhere

[38] D. E. O'Leary, "Artificial intelligence and big data," *IEEE Intell. Syst.*, vol. 28, no. 2, pp. 96–99, Mar. 2013.

[39] Y. Demchenko, P. Grosso, C. de Laat, and P. Membrey, "Addressing big data issues in scientific data infrastructure," in *Proc. Int. Conf. Collaboration Technol. Syst. (CTS)*, May 2013, pp. 48–55.

[40] S. Mittal and J. S. Vetter, "A survey of CPU-GPU heterogeneous computing techniques," *ACM Comput. Surv.*, vol. 47, no. 4, pp. 69:1–69:35, Jul. 2015. [Online]. Available: http://doi.acm.org/10.1145/2788396

[41] D. Singh and C. K. Reddy, "A survey on platforms for big data analytics," *J. Big Data*, vol. 2, no. 1, p. 8, Dec. 2015. [Online]. Available: https://link.springer.com/article/10.1186/s40537-014-0008-6

[42] X. Tian, R. Han, L. Wang, G. Lu, and J. Zhan, "Latency critical big data computing in finance," *J. Finance Data Sci.*, vol. 1, no. 1, pp. 33–41, Dec. 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2405918815000045

[43] L. Djafri, D. Amar Bensaber, and R. Adjoudj, "Big data analytics for prediction: Parallel processing of the big learning base with the possibility of improving the final result of the prediction," *Inf. Discovery Del.*, vol. 46, no. 3, pp. 147–160, Aug. 2018. [Online]. Available: https://www.emeraldinsight.com/doi/10.1108/IDD-02-2018-0002

[44] Y. Etsion and D. Tsafrir, "A short survey of commercial cluster batch schedulers," School Comput. Sci. Eng., Hebrew Univ. Jerusalem, Jerusalem, Israel, vol. 44221, 2005, p. 13.

[45] L. Sliwko and V. Getov, "Workload schedulers-genesis, algorithms and comparisons," *Int. J. Comput. Sci. Softw. Eng.*, vol. 4, no. 6, pp. 141–155, 2015.

[46] M. S. Khaira, "Fast first-come first served arbitration method," U.S. Patent 5 574 867 A, Nov. 12, 1996. [Online]. Available: http://www.google.com/patents/US5574867

[47] D. Jackson, Q. Snell, and M. Clement, "Core Algorithms of the Maui Scheduler," in *Job Scheduling Strategies for Parallel Processing* (Lecture Notes in Computer Science). Berlin, Germany: Springer, Jun. 2001, pp. 87–102. [Online]. Available: https://link.springer.com/chapter/10.1007/3-540-45540-X_6

[48] (Nov. 2017). *Apache Hadoop 2.9.0—Hadoop: Capacity Scheduler*. [Online]. Available: https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html

[49] *Apache Hadoop 2.9.0—Hadoop: Fair Scheduler*. Accessed: Nov. 2017. [Online]. Available: https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/FairScheduler.html

[50] T. Tannenbaum, D. Wright, K. Miller, and M. Livny, "Condor: A distributed job scheduler," in *Beowulf Cluster Computing With Linux*, T. Sterling, Ed. Cambridge, MA, USA: MIT Press, Oct. 2001.

[51] *HTCondor—Research Publications and Technical Information*. Accessed: Jan. 18, 2021. [Online]. Available: https://research.cs.wisc.edu/htcondor/publications.html

[52] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proc. IEEE 26th Symp. (MSST)*, May 2010, pp. 1–10.

[53] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "SCOPE: Easy and efficient parallel processing of massive data sets," *Proc. VLDB Endowment*, vol. 1, no. 2, pp. 1265–1276, Aug. 2008.

[54] F. B. Schmuck and R. L. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proc. FAST*, vol. 2, 2002, pp. 1–14.

[55] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant Web services," *ACM SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002, doi: 10.1145/564585.564601.

[56] B. Liskov, L. Shrira, and J. Wroclawski, "Efficient at-most-once messages based on synchronized clocks," *ACM Trans. Comput. Syst.*, vol. 9, no. 2, pp. 125–142, May 1991, doi: 10.1145/103720.103722.

[57] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm Twitter," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*. New York, NY, USA: ACM, 2014, pp. 147–156. [Online]. Available: http://doi.acm.org/10.1145/2588555.2595641

[58] Y. Huang and H. Garcia-Molina, "Exactly-once semantics in a replicated messaging system," in *Proc. 17th Int. Conf. Data Eng.*, 2001, pp. 3–12.

[59] F. Junqueira. *Making Sense of Exactly-Once Semantic*. Strate. London, U.K. Accessed: 2016. [Online]. Available: https://cdn.oreillystatic.com/en/assets/1/event/155/Making%20sense%20of%20exactly-once%20semantics%20Presentation%201.pdf

[60] *Apache Flink 1.3 Documentation: Distributed Runtime Environment*. Accessed: Dec. 2017. [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.3/concepts/runtime.html

[61] *Apache Hadoop—2.9.0 Using CGroups With YARN*. Accessed: Nov. 2017. [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/NodeManagerCgroups.html

[62] *Cluster Mode Overview—Spark 2.2.0 Documentation*. Accessed: Jan. 18, 2021. [Online]. Available: https://spark.apache.org/docs/latest/cluster-overview.html

[63] *Samza*. Accessed: Jan. 18, 2021. [Online]. Available: http://samza.apache.org/

[64] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Conf. Hot Topics Cloud Comput. (HotCloud)*. Berkeley, CA, USA: USENIX Association, 2010, p. 10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113

[65] D. J. Becker, T. Sterling, D. Savarese, J. E. Dorband, U. A. Ranawak, and C. V. Packer, "BEOWULF: A parallel workstation for scientific computation," in *Proc. Int. Conf. Parallel Process.*, vol. 95, 1995, pp. 1–9.

[66] V. S. Sunderam, "PVM: A framework for parallel distributed computing," *Concurrency, Pract. Exper.*, vol. 2, no. 4, pp. 315–339, Dec. 1990.

[67] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proc. 11th Eur. PVM/MPI Users Group Meeting*, Budapest, Hungary, Sep. 2004, pp. 97–104.

[68] M. Schulz, "MPI: A message-passing interface standard version 3.1," Message Passing Interface Forum, Chicago, IL, USA, Tech. Rep., Apr. 2016.

[69] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. 6th Symp. Operating Syst. Design Implement.*, vol. 51, 2004, p. 107.

[70] M. Hausenblas and N. Bijnens. *Lambda Architecture*. Accessed: 2017. [Online]. Available: http://lambda-architecture.net/

[71] Z. Hasani, M. Kon-Popovska, and G. Velinov, "Lambda architecture for real time big data analytic," in *Proc. ICT Innov. Web*, 2014, pp. 133–143.

[72] N. Marz and J. Warren, *Big Data: Principles and Best Practices of Scalable Real-time Data Systems*, 1st ed. Shelter Island, NY, USA: Manning Publications, 2015. [Online]. Available: https://books.google.co.uk/books?id=HW-kMQEACAAJ

[73] O. Boykin, S. Ritchie, I. O'Connell, and J. Lin, "Summingbird: A framework for integrating batch and online MapReduce computations," *Proc. VLDB Endowment*, vol. 7, no. 13, pp. 1441–1451, Aug. 2014, doi: 10.14778/2733004.2733016.

[74] J. Kreps. *Questioning the Lambda Architecture*. Accessed: Jul. 2014. [Online]. Available: https://www.oreilly.com/ideas/questioning-the-lambda-architecture

[75] M. Kleppmann. *Turning the Database Inside Out With Apache Samza*. Strangeloop. Accessed: 2014. [Online]. Available: https://www.youtube.com/watch?v=fU9hR3kiOK0

[76] The Apache Software Foundation. *Welcome to Apache Hadoop!* Accessed: 2017. [Online]. Available: http://hadoop.apache.org/

[77] K. V. Shvachko, "HDFS scalability: The limits to growth," *Login, Mag. USENIX SAGE*, vol. 35, no. 2, pp. 6–16, Jan. 2010. [Online]. Available: http://c59951.r51.cf2.rackcdn.com/5424-1908-shvachko.pdf

[78] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Conf. Netw. Syst. Design Implement.* Berkeley, CA, USA: USENIX Association, 2012, p. 2.

[79] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proc. ACM Symp. Operating Syst. Princ.* New York, NY, USA: ACM, 2013, pp. 423–438. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2517349.2522737

[80] The Apache Software Foundation. *Apache Storm*. Accessed: 2015. [Online]. Available: http://storm.apache.org/

[81] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bull. IEEE Comput. Soc. Tech. Committee Data Eng.*, vol. 36, no. 4, pp. 1–12, 2015. [Online]. Available: http://kth.diva-portal.org/smash/record.jsf?pid=diva2:1059537

[82] The Apache Software Foundation. *Apache Flink: Scalable Stream and Batch Data Processing*. Accessed: 2017. [Online]. Available: https://flink.apache.org/

[83] The Apache Software Foundation. *Apache Hadoop 2.9.0—Apache Hadoop YARN*. Accessed: Nov. 2017. [Online]. Available: http://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html

[84] V. K. Vavilapalli, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, E. Baldeschwieler, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, and H. Shah, "Apache Hadoop YARN: Yet another resource negotiator," in *Proc. SOCC*. New York, NY, USA: ACM, 2013, pp. 1–16. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2523616.2523633

[85] The Apache Software Foundation. *Apache Twill—Home*. Accessed: 2017. [Online]. Available: http://twill.apache.org/

[86] The Apache Software Foundation. *Apache ZooKeeper—Home*. Accessed: 2017. [Online]. Available: https://zookeeper.apache.org/

[87] The Apache Software Foundation. *Apache Curator*. Accessed: Dec. 2018. [Online]. Available: https://curator.apache.org/

[88] *HTCondor—What is HTCondor?* Accessed: Jan. 18, 2021. [Online]. Available: http://research.cs.wisc.edu/htcondor/description.html

[89] S. Hoffman, *Apache Flume: Distributed Log Collection for Hadoop*. Birmingham, U.K.: Packt, Jan. 2013.

[90] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A distributed messaging system for log processing," in *Proc. NetDB*, 2011, pp. 1–7.

[91] Public Services. *Documentation: Table of Contents—RabbitMQ*. Accessed: Jul. 2019. [Online]. Available: https://www.rabbitmq.com/documentation.html

[92] I. Peattie. *The Bench Mark Database*. Accessed: Jan. 18, 2021. [Online]. Available: http://www.bench-marks.org.uk/

[93] *DB-Engines—Knowledge Base of Relational and NoSQL Database Management Systems*. Accessed: Jan. 18, 2021. [Online]. Available: https://db-engines.com/en/

[94] A. Ashok, "Four trends in cloud computing CIOs should prepare for in 2019," Forbes, Jersey City, NJ, USA, Tech. Rep., 2019. [Online]. Available: https://www.forbes.com/sites/forbestechcouncil/2018/07/05/four-trends-in-cloud-computing-cios-should-prepare-for-in-2019/

[95] J. Cao, K. Hwang, K. Li, and A. Y. Zomaya, "Optimal multiserver configuration for profit maximization in cloud computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 6, pp. 1087–1096, Jun. 2013.

[96] J. Bi, H. Yuan, W. Tan, M. Zhou, Y. Fan, J. Zhang, and J. Li, "Application-aware dynamic fine-grained resource provisioning in a virtualized cloud data center," *IEEE Trans. Autom. Sci. Eng.*, vol. 14, no. 2, pp. 1172–1184, Apr. 2017.

[97] C. Metz, "Google's Dremel makes big data look small," Wired, Boone, IA, USA, Tech. Rep., Aug. 2012.

[98] C. Metz, "How Facebook knows what you really like," Wired, Boone, IA, USA, Tech. Rep., May 2012.

[99] H. Yuan, J. Bi, W. Tan, M. Zhou, B. H. Li, and J. Li, "TTSA: An effective scheduling approach for delay bounded tasks in hybrid clouds," *IEEE Trans. Cybern.*, vol. 47, no. 11, pp. 3658–3668, Nov. 2017.

[100] F. Zhang, J. Cao, K. Hwang, K. Li, and S. U. Khan, "Adaptive workflow scheduling on cloud computing platforms with IterativeOrdinal optimization," *IEEE Trans. Cloud Comput.*, vol. 3, no. 2, pp. 156–168, Apr. 2015.

[101] E. Strubell, A. Ganesh, and A. McCallum, "Energy and policy considerations for deep learning in NLP," in *Proc. 57th Annu. Meeting Assoc. Comput. Linguistics*. Florence, Italy: Association for Computational Linguistics, Jul. 2019, pp. 3645–3650.

[102] H. Yuan, H. Liu, J. Bi, and M. Zhou, "Revenue and energy cost-optimized biobjective task scheduling for green cloud data centers," *IEEE Trans. Autom. Sci. Eng.*, early access, Feb. 25, 2020, doi: 10.1109/TASE.2020.2971512.

[103] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: Interactive analysis of Web-scale datasets," Google, Mountain View, CA, USA, Tech. Rep., 2010, p. 10.
[104] *BigQuery Omni for Multi-Cloud Data Analytics*. Accessed: Jan. 18, 2021. [Online]. Available: https://cloud.google.com/blog/products/data-analytics/introducing-bigquery-omni/
[105] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proc. 21st ACM SIGMOD-SIGACT-SIGART Symp. Princ. Database Syst. (PODS)*. New York, NY, USA: ACM, 2002, pp. 1–16, doi: 10.1145/543613.543615.
[106] M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining stream statistics over sliding windows," in *Proc. 13th Annu. ACM-SIAM Symp. Discrete Algorithms (SODA)*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002, pp. 635–644. [Online]. Available: http://dl.acm.org/citation.cfm?id=545381.545466
[107] J. Han, J. Pei, and M. Kamber, *Data Mining: Concepts and Techniques*. Amsterdam, The Netherlands: Elsevier, 2011.
[108] A. Bifet and R. Gavalda, "Learning from time-changing data with adaptive windowing," in *Proc. SIAM Int. Conf. Data Mining*. Philadelphia, PA, USA: SIAM, 2007, pp. 443–448.
[109] E. S. Page, "Continuous inspection schemes," *Biometrika*, vol. 41, nos. 1–2, pp. 100–115, 1954.
[110] J. Gama, P. Medas, G. Castillo, and P. Rodrigues, "Learning with drift detection," in *Proc. Brazilian Symp. Artif. Intell.* Berlin, Germany: Springer, 2004, pp. 286–295.
[111] P. B. Dongre and L. G. Malik, "Stream data classification and adapting to gradual concept drift," *Int. J.*, vol. 2, no. 3, pp. 1–5, 2014.
[112] M. Baena-García, J. del Campo-Ávila, R. Fidalgo, A. Bifet, R. Gavaldà, and R. Morales-Bueno, "Early drift detection method," in *Proc. Int. Workshop Knowl. Discovery Data Streams*, 2006, pp. 77–86.
[113] Z. Yang, S. Al-Dahidi, P. Baraldi, E. Zio, and L. Montelatici, "A novel concept drift detection method for incremental learning in nonstationary environments," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 31, no. 1, pp. 309–320, Jan. 2020.
[114] G. Hulten and P. Domingos, "VFML—A toolkit for mining high-speed time-changing data streams," *Softw. Toolkit*, vol. 51, p. 51, Oct. 2003.
[115] G. Widmer and M. Kubat, "Learning in the presence of concept drift and hidden contexts," *Mach. Learn.*, vol. 23, no. 1, pp. 69–101, Apr. 1996.
[116] G. Hulten, L. Spencer, and P. Domingos, "Mining time-changing data streams," in *Proc. 7th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD)*, 2001, pp. 97–106.
[117] J. Gama and P. Kosina, "Learning decision rules from data streams," in *Proc. Int. Joint Conf. Artif. Intell. (IJCAI)*, 2011, vol. 22, no. 1, p. 1255.
[118] T. Le, F. Stahl, M. M. Gaber, J. B. Gomes, and G. D. Fatta, "On expressiveness and uncertainty awareness in rule-based classification for data streams," *Neurocomputing*, vol. 265, pp. 127–141, Nov. 2017. http://www.sciencedirect.com/science/article/pii/S0925231217310172
[119] S. Fong, R. Wong, and A. V. Vasilakos, "Accelerated PSO swarm search feature selection for data stream mining big data," *IEEE Trans. Services Comput.*, vol. 9, no. 1, pp. 33–45, Feb. 2016.
[120] J. Shao, Z. Ahmadi, and S. Kramer, "Prototype-based learning on concept-drifting data streams," in *Proc. 20th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD)*, 2014, pp. 412–421.
[121] J. P. Barddal, H. M. Gomes, and F. Enembreck, "SFNClassifier: A scale-free social network method to handle concept drift," in *Proc. 29th Annu. ACM Symp. Appl. Comput. (SAC)*, 2014, pp. 786–791.
[122] H. Ghomeshi, M. M. Gaber, and Y. Kovalchuk, "RED-GENE: An evolutionary game theoretic approach to adaptive data stream classification," *IEEE Access*, vol. 7, pp. 173944–173954, 2019.
[123] T. Zhang, R. Ramakrishnan, and M. Livny, "BIRCH: An efficient data clustering method for very large databases," *ACM Sigmod Record*, vol. 25, no. 2, pp. 103–114, Jun. 1996.
[124] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu, "A framework for clustering evolving data streams," in *Proc. 29th Int. Conf. Very Large Data Bases (VLDB)*, vol. 29. VLDB Endowment, 2003, pp. 81–92.
[125] F. Cao, M. Estert, W. Qian, and A. Zhou, "Density-based clustering over an evolving data stream with noise," in *Proc. SIAM Int. Conf. Data Mining*. Philadelphia, PA, USA: SIAM, Apr. 2006, pp. 328–339.
[126] S. Xu, L. Feng, S. Liu, and H. Qiao, "Self-adaption neighborhood density clustering method for mixed data stream with concept drift," *Eng. Appl. Artif. Intell.*, vol. 89, Mar. 2020, Art. no. 103451.
[127] M. Tennant, F. Stahl, O. Rana, and J. B. Gomes, "Scalable real-time classification of data streams with concept drift," *Future Gener. Comput. Syst.*, vol. 75, pp. 187–199, Oct. 2017.
[128] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct. 2016.
[129] M. Polese, R. Jana, V. Kounev, K. Zhang, S. Deb, and M. Zorzi, "Machine learning at the edge: A data-driven architecture with applications to 5G cellular networks," *IEEE Trans. Mobile Comput.*, early access, Jun. 3, 2020, doi: 10.1109/TMC.2020.2999852.
[130] M. M. Gaber, J. B. Gomes, and F. Stahl, "Pocket data mining," in *Big Data on Small Devices* (Studies in Big Data). Zürich, Switzerland: Springer, 2014.
[131] H. Kargupta, B.-H. Park, S. Pittie, L. Liu, D. Kushraj, and K. Sarkar, "MobiMine: Monitoring the stock market from a PDA," *ACM SIGKDD Explorations Newslett.*, vol. 3, no. 2, pp. 37–46, Jan. 2002.
[132] H. Kargupta, R. Bhargava, K. Liu, M. Powers, P. Blair, S. Bushra, J. Dull, K. Sarkar, M. Klein, M. Vasa, and D. Handy, "VEDAS: A mobile and distributed data stream mining system for real-time vehicle monitoring," in *Proc. SIAM Int. Conf. Data Mining*. Philadelphia, PA, USA: SIAM, Apr. 2004, pp. 300–311.
[133] H. Kargupta, V. Puttagunta, M. Klein, and K. Sarkar, "On-board vehicle data stream monitoring using MineFleet and fast resource constrained monitoring of correlation matrices," *New Gener. Comput.*, vol. 25, no. 1, pp. 5–32, Nov. 2006.
[134] P. D. Haghighi, S. Krishnaswamy, A. Zaslavsky, M. M. Gaber, A. Sinha, and B. Gillick, "Open mobile miner: A toolkit for building situation-aware data mining applications," *J. Organizational Comput. Electron. Commerce*, vol. 23, no. 3, pp. 224–248, Jul. 2013.
[135] F. Stahl, M. M. Gaber, M. Bramer, and P. S. Yu, "Pocket data mining: Towards collaborative data mining in mobile computing environments," in *Proc. 22nd IEEE Int. Conf. Tools Artif. Intell.*, vol. 2, Oct. 2010, pp. 323–330.
[136] P. Prakash Jayaraman, C. Perera, D. Georgakopoulos, and A. Zaslavsky, "MOSDEN: A scalable mobile collaborative platform for opportunistic sensing applications," 2014, *arXiv:1405.5867*. [Online]. Available: http://arxiv.org/abs/1405.5867
[137] M. Habib ur Rehman, C. S. Liew, and T. Y. Wah, "UniMiner: Towards a unified framework for data mining," in *Proc. 4th World Congr. Inf. Commun. Technol. (WICT)*, Dec. 2014, pp. 134–139.
[138] M. Habib ur Rehman, P. P. Jayaraman, S. U. R. Malik, A. U. R. Khan, and M. M. Gaber, "Rededge: A novel architecture for big data processing in mobile edge computing environments," *J. Sensor Actuator Netw.*, vol. 6, no. 3, p. 17, 2017.
[139] T. Mahmood and U. Afzal, "Security analytics: Big data analytics for cybersecurity," in *Proc. 2nd Nat. Conf. Inf. Assurance (NCIA)*, 2013, pp. 129–134.
[140] M. M. Hossain, M. Fotouhi, and R. Hasan, "Towards an analysis of security issues, challenges, and open problems in the Internet of Things," in *Proc. IEEE World Congr. Services*, Jun. 2015, pp. 21–28. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7196499
[141] S. Bi, R. Zhang, Z. Ding, and S. Cui, "Wireless communications in the era of big data," *IEEE Commun. Mag.*, vol. 53, no. 10, pp. 190–199, Oct. 2015.

TIMOTHÉE DUBUC received the M.Eng. degree from SupInfo/IngeSup Engineer Schools, France, the M.Sc. degree in machine learning from the Faculty of Science, Orsay, France, and the Ph.D. degree in cybernetics from the University of Reading, U.K.

Since 2007, his work has interweaved industrial and academic goals, to ensure insights from neuroscience, and the way the human nervous system processes information, can be used to inform the development of applications in artificial intelligence and machine learning. In this career, he has been an active Developer of data infrastructures, visualization platforms, and game engines. His Ph.D. work involved the modeling of the neural architecture of cells in the human retina to make apparent the mechanisms that give rise to low-level features of perception like the detection of edges in a visual scene. This work further aims to inform the design and application of artificial neural networks.

FREDERIC STAHL received the Dipl.Ing. (FH) degree in bioinformatics from the University of Applied Sciences, Weihenstephan, Germany, in 2006, and the Ph.D. degree in computer science from the University of Portsmouth, U.K., in 2010.

From 2010 to 2012, he was a Senior Research Associate with the Department of Computer Science, University of Portsmouth. In 2012, he has worked as a Lecturer with the Department of Design Engineering and Computing, Bournemouth University, U.K. From 2012 to 2019, he was a Lecturer and an Associate Professor with the University of Reading, U.K. Since 2019, he has been the Deputy Head, the Team Leader, and a Senior Researcher for marine perception with the German Research Centre for Artificial Intelligence (DFKI GmbH). He has published more than 60 articles in peer-reviewed conferences, journals, and book chapters. He has been working in the field of data mining for more than ten years focusing on the research domain of big data analytics. His particular research interests include developing scalable algorithms for building adaptive models for real-time streaming data, developing scalable parallel data mining algorithms and workflows, and applications in big data analytics.

Dr. Stahl is a member of the British Computer Society (BCS) and has been elected three times as a Committee Member of the BCS's Specialist Group on Artificial Intelligence (SGAI), serving on the committee, since 2013.

ETIENNE B. ROESCH received the B.Sc. degree in software engineering and systems integration, the B.Sc. and M.Sc. degrees in cognitive science, and the Ph.D. degree in psychology.

He held positions at Harvard University, the University of Geneva, the Imperial College London, Goldsmiths University London, and the University of Reading, U.K., where he is currently an Associate Professor in cognitive science. His research interests include interdisciplinary, and focuses on exploring the interface between technology and neuroscience, to understand human cognition and inform the development of future and emerging technology. His research laboratory led projects related to human perception and consciousness, sensory augmentation, brain–computer interfaces, the Internet of Things, as well as the development of novel statistical methods for the analysis of concurrent EEG-fMRI, and machine learning. Recently, he has been coordinating the EPSRC project Cocoon, which combines cyber security with emotion psychology, and served on the steering board of the EU Network of Excellence HUMAINE. The latter led to two handbooks in affective computing. He is also the Deputy Director of the Centre for Integrative Neuroscience and Neurodynamics, and manages the MRI, EEG, and TMS facilities.

Dr. Roesch is a member of the Emerging Applications Section of the Royal Statistical Society, a local representative of the U.K. Reproducibility Network, and the Founder and the Chief Editor of the journal *ReScience X*, dedicated to the publication of reproductions and replications of experimental work.