Int J Parallel Prog (2014) 42:710–738
DOI 10.1007/s10766-013-0272-7

Parallel Programming Paradigms and Frameworks in Big Data Era

Ciprian Dobre · Fatos Xhafa

Received: 18 July 2013 / Accepted: 20 August 2013 / Published online: 1 September 2013

Abstract With Cloud Computing emerging as a promising new approach for ad-hoc parallel data processing, major companies have started to integrate frameworks for parallel data processing in their product portfolio, making it easy for customers to access these services and to deploy their programs. We have entered the Era of Big Data. The explosion and profusion of available data in a wide range of application domains rise up new challenges and opportunities in a plethora of disciplines—ranging from science and engineering to biology and business. One major challenge is how to take advantage of the unprecedented scale of data—typically of heterogeneous nature—in order to acquire further insights and knowledge for improving the quality of the offered services. To exploit this new resource, we need to scale up and scale out both our infrastructures and standard techniques. Our society is already data-rich, but the question remains whether or not we have the conceptual tools to handle it. In this paper we discuss and analyze opportunities and challenges for efficient parallel data processing. Big Data is the next frontier for innovation, competition, and productivity, and many solutions continue to appear, partly supported by the considerable enthusiasm around the MapReduce paradigm for large-scale data analysis. We review various parallel and distributed programming paradigms, analyzing how they fit into the Big Data era, and present modern emerging paradigms and frameworks. To better support

This work was supported by project "ERRIC -Empowering Romanian Research on Intelligent Information Technologies/FP7-REGPOT-2010-1", ID: 264207.

C. Dobre (✉)
Computer Science Department, University Politehnica of Bucharest,
Spl. Independentei 313, Bucharest, Romania
e-mail: ciprian.dobre@cs.pub.ro

F. Xhafa
Universitat Politecnica de Catalunya, Girona Salgado 1-3, 08034 Barcelona, Spain
e-mail: fatos@lsi.upc.edu

practitioners interesting in this domain, we end with an analysis of on-going research challenges towards the truly fourth generation data-intensive science.

Keywords Parallel programming · Big Data · MapReduce · Programming models · Challenges

1 Introduction

Today the Internet represents a big space where great amounts of information are added every day. IBM claims 90 % of the world's data has been accumulated only since 2010 [27]. Large datasets of information is indisputable being amassed as a result of our social, mobile, and digital world. Since 2012, the use of the term in the U.S. has increased 1,211 % on the Internet [13]. We are not far from the time when terms like PetaByte, ExaByte, and ZettaByte[1] will be quite common [4]. Such amount of information might already be there, but we just don't know it.[2] Part of the sum is also the annual global IP traffic (0.8 ZettaByte) and annual Internet Video (0.3 ZettaByte) as we entered 2013 [5].

There are various mechanisms that generate Big Data: data from scientific measurements and experiments (astronomy, physics, genetics, etc.), peer-to-peer communication (text messaging, chat lines, digital phone calls), broadcasting (News, blogs), social networking (Facebook, Twitter), authorship (digital books, magazines, Web pages, images, videos), administrative data (enterprise or government documents, legal and financial records), business data (e-commerce, stock markets, business intelligence, marketing, advertising). Healthcare data are being processed at incredible speed, putting in data from a number of sources to turn into rapid insights. Insurance, retail and Web companies are also tapping into big, fast data based on transactions. Life sciences companies and research entities alike are turning to new Big Data technologies to enhance their results from their high performance systems by allowing more complex data ingestion and processing. E-commerce websites and social networks, also cope with enormous volumes of data. Such services generate clickstream data from millions of users every day, which is a potential gold mine for understanding access patterns and increasing ad revenue.

But, in the Big Data era, storing huge amounts data is not the biggest challenge anymore. Companies already store huge amounts of information. As an example, Facebook is able to store activity data in its back-end ranging all the way back to 2005, when the company was just two years old [36]. The company only deletes data when it is required to do so for security, privacy, or regulatory reasons. Appart from storing, today researchers struggle with designing solutions to *understand* this

[1] To understand the complexity in working with such amounts of data, think of what would happen if someone accidentally pushes the Print button and 1 ZettaByte of data would be printed on paper. Actually, this amount of printed information would weigh about 1,016 pounds or $5 \times 1{,}010$ tonnes. One ZettaByte of equivalent books would fill up 10 billion Trucks or 500,000 aircraft carriers, and if equally distributed they would mean 10,000 books for each person living on the planet today. To make just the paper to print on would require 3 times the number of trees in the world today [4].

[2] Various experts predict that the World Wide Web might already contain 1 ZettaByte of information.

Big amount of *Data*. Efficient parallel and concurrent algorithms and implementation techniques are needed to meet the scalability and performance requirements entailed by scientific data analyses. Challenges such as *scalability* and *resiliance to failure* are already being addressed at the infrastructure layer. But new Big Data problems relate to users *handling too many files*, and/or working with *very large files*. Applications need *fast movement and operations* on that data, not to mention support to cope with an *incredible diversity of data*. Big Data issues also emerge from *extensive data sharing*, allowing multiple users to explore or analyze the same data set. All these demand a new movement and a new set of complementary technologies. Big Data is the new way to process and analyze existing data and new data sources. Up until recently Google, Amazon and Microsoft were the main actors capable to handle Big Data, but today new actors enter the Big Data stage. At the forefront, Google uses one of the largest dataset to discover interesting aspects to improve their services. MapReduce, their novel computing model, provoked the scientists to rethink how large-scale data operations should be handled. Today Google introduces novel Big Data tools, such as BigQuery, that appeal to many more users. Yahoo's Hadoop, Microsoft's Driad, are other examples of powerful computing tools that led to the new paradigm shift called "Big Data". Hadoop, for example, spreads data across a sea of commodity servers, before using the collective power of those machines to transform the data into something useful. It is highly attractive because commodity servers are cheap, and as your data expands, you just add more of them. But it is certainly not the only available solution out there.

In this paper we discuss and analyze opportunities and challenges for efficient parallel data processing. We review various parallel and distributed programing paradigms, analyzing how they fit into the Big Data era, and present modern emerging paradigms and frameworks.

The rest of this paper is organized as follows. Section 2 presents a short history of Big Data paradigms. This is followed in Sect. 3 by an analysis of current programing models and technologies tailored specifically for the Big Data era. In Sect. 4 we present a series of challenges that were identified in the previous analysis. Section 5 concludes the paper.

2 A History of Big Data Paradigms

Several steps preluded the Big Data era. Back in the 90s, the data volumes generated (mainly) by companies was sufficiently low that the database management system itself would figure out the best access path to the data via an *optimizer*. With the rise in the opportunities coming from the transaction-oriented market, within a few years the database world became quite a competitive place: Ingres, Informix, Sybase and Oracle battled along IBM in the enterprise transaction process market. A gradual awareness that such databases were all optimized for transaction processing performance allowed a further range of innovation, and the first *specialist analytical databases* appeared (i.e., we mention here pioneer products such as Red Brick, Essbase). This storm of innovation was soon coming to a pace, as by the dawn of the millennium the database market had seen dramatic consolidation: Oracle, IBM and Microsoft dominated the landscape, having either bought out or crushed most of the competition. Object

databases came and go, and the database administrator beginning his career at that time could look forward to a stable world of a few databases, all based on SQL. Few appreciated at the time the rapid growth in both the volume and types of data that companies collect was about to challenge the database incumbents and spawn another round of innovation.

Whilst Moore's Law was holding for processing speed, it was most decidedly not working for disk access speed (even if the density of information a hard disk can record has grown about 50 million times over the past 30 years). Solid-state drives helped, but they were (and still are) quite expensive. Database volumes were increasing faster than ever, due primarily to the explosion of social media data and machine-generated data, such as information from sensors, point-of-sale systems, mobile phone network, Web server logs and the like. The data explosion soon was indisputable recognized to be the main driver of contemporary innovation: a fourth paradigm of scientific knowledge generation was born!

We arguably recognize today three main approaches to generating new knowledge: experimental and theoretical researches are classified as the first two, while more recently computer simulations of natural phenomena (and of engineering artefacts) have contributed a third. In fact, Bell et al. [3] have proposed a fourth—*data-intensive science*. Like any other major shift in scientific thinking, *data-intensive science* both represents and is driven by a change in the scientific landscape. It is not just a re-statement of the significance of data-driven rather than hypothesis-dependent science. In this case, it is the ability of modern instrumentation to generate data at rates 100–1,000-fold that of the devices they are replacing. In 1997 the largest commercial database in the world was 7 TB in size, and that figure had only grown to about 30 TB by 2003 [16]. Yet, it more than tripled to 100 TB by 2005, and by 2008 the first petabyte-sized database appeared. In other words, the largest databases increased tenfold in size between 2005 and 2008. The strains of analyzing such volumes of data started to stretch and exceed the capacity of the mainstream databases.

The database industry has responded differently. Massively parallel processing (MPP) databases allow database loads to be split amongst many processors. The columnar data structure pioneered by Sybase turned out to be well suited to analytical processing workloads, and a range of new analytical databases soon followed, often combining columnar and MPP approaches. The big database vendors responded with either their own versions, or by simply purchasing upstart rivals. For example, Oracle brought its Exadata offering, IBM purchased Netezza and Microsoft bought DATAllegro.

Things took a different turn when dynamic computations over ever larger amounts of data became a necessity. SQL became inapt for the challenge. Big Data workflows were needed more and more, involving transformations of large amounts of information, often unstructured, into data science driven insights, or highly available data stores used by applications. Google, having to deal with exponentially growing Web traffic, coped with this by devising its own approach called MapReduce, designed to work with distributed processing in massively distributed file systems. That work inspired an open source technology called Hadoop, along with an associated file system called HDFS. Databases followed the same trend, endeavoring to allow more predictable scalability and eliminating the constraints of the until-then fixed database

schema. NoSQL database soon appeared, and Big Data challenged more traditional mechanisms, which leaded in the last years to new distributed processing paradigms: Pig, Sawzall, Dryad, and many others. The Big Data era was born!

The problem with Big Data workflows is that one has to define several tasks required to accomplish the needed transformations. In particular, state is considered the biggest enemy of dynamic computations. Although it cannot be eliminated, it can be avoided by defining data driven workflows in terms of functions, predicates, and tuples (as opposed to defining a workflow in terms of sequential steps). For example, let's say we want to take a list of conversations, extract human entities (names) from them, and output a list of human entities who appear to be linked to one another (i.e. two users are linked if they have had a recent conversation). If we broke this into a linear flow of tasks, it might take more than a day to process all this information if we have, say, a petabyte of raw text sitting on a computation cluster of several standard machines. We can define these transformations imperatively, using a low-level framework like Hadoop's MapReduce. Alternatively, we can define it using flows (i.e. using a framework such as Cascading/Cascalog, which is based on a declarative programming paradigm). This is why functional programming (FP) is actually considered today to be the most prominent Programming Paradigm, as it allows actually more flexibility in defining Big Data distributed processing workflows. FP is adequate for data problems, since it emphasizes the abstractions that are most appropriate for data analysis. In fact, SQL can be considered a functional programming language, since it is derived from Set Theory, although it has lots of limitations as a language. Object-oriented Programming, on the other hand, does not necessarily support well mathematical abstractions needed for data analysis: the main reason why Java in Big Data applications is considered by many to be counterproductive. There are hybrid programming languages, such as Scala, F#, and OCaml, which combine both FP with OOP paradigms.

There are two other emerging trends in programming that will probably impact the data world of tomorrow [42]. Logic programming (LP), like FP, is actually not that new, but is seeing a resurgence of interest, especially in the Clojure community. Rules engines, like Drools, are an LP category that has been in use for a long time. In LP, one writes programs using the concepts of Logic, such as first order logic. Simply stated, one specifies conditions or constraints (e.g., rules) that must be satisfied, known "facts" about the system being modeling, and the runtime automatically finds the values of the system's variables that satisfy the conditions. In other words, the runtime searches the space of all possible answers for those that satisfy the conditions and facts. If a problem fits the LP model, one can work quickly and efficiently, in just the same way that SQL queries are a very concise and expressive way to ask questions of data and to perform analytics.

This is quite interesting for Big Data analysis as well. For example, a classic use of LP has been fault diagnosis: given observed events or symptoms and knowledge of the system, researchers are trying to automatically detect what are the possible underlying faults that caused the observations [14]. The problem is that most LP systems assume absolute knowledge: facts are yes/no, while constraints are absolute and comprehensive. However, many real-world scenarios are not so clear cut. Even so, today we witness several examples where LP is already successful used for Big Data analysis. Recommendation engines are widely used in social networks and e-

commerce. For example, Netflix is able to observe when a user rents action movies more often than romantic comedies and make intelligent recommendation accordingly.

Probabilistic modeling is another trend that has proven fruitful for scenarios where knowledge and constraints are imprecise and contain gaps. For Netflix, using only LP the analyzing system is not able to classify correctly, let's say, romantic comedies with car chases. For self-navigating robots, who work by using an internal model of the world (e.g., a map of the terrain and sensors used to detect where they are), a Big Data problem is that they have to analyze large amounts of data that is produced by sources prone to error and uncertainty. Real sensors are not 100 % accurate. The map could have errors and obstacles could be in the way (like people crossing the street) but not represented on the map. So, the world has to be modeled probabilistically, and the robot calculates the most likely location, given its measurements and how they correlate to the map. Google's self-driving car is an example of a beautiful application of exactly this principle for coping with large error-prone collections of navigation data [24]. Other Smart City applications in this category will soon follow [29].

Thus, today we already witness an explosion of applications of powerful probabilistic modeling techniques and tools, such as Bayesian networks, Markov networks, and their variants, generically called Probabilistic Graphical Models (because they model probabilities about systems using graphs) to Big Data problems. Implementations are available in many languages. However, deeper technical expertise is required to understand and use these techniques effectively. We're now on the verge of moving to the next level, probabilistic programming languages and systems that make it easier to build probabilistic models, where the modeling concepts are promoted to first-class primitives in new languages, with underlying runtimes that do the hard work of inferring answers, similar to the way that logic programming languages work already. The ultimate goal is to enable end users with limited programming skills, like domain experts, to build effective probabilistic models, without requiring the assistance of Ph.D.-level machine learning experts, much the way that SQL is widely used today. DARPA, the research arm of the US Department of Defense, considers this trend important enough that they are starting an initiative to promote it, called Probabilistic Programming for Advanced Machine Learning [2]. This is a next logical step in the *democratization of data*, making the sophisticated analysis of large data sets accessible to a wider audience. The world is advancing towards an era where SQL knowledge will be universally accessible even to very nontechnical people who will have to learn just enough basic SQL to get the answers they need for themselves. But, until then, let's see what programming models revolve around the Big Data hype today.

3 Programming Models for Big Data Era

Today's sheer volume of data that Internet services work with has led to interest in parallel processing on commodity hardware. There is a sense among some Big Data leaders that the infrastructure challenge has largely been met. But as the volumes of data swell into exabyte territory for more and more organizations, the biggest challenge is going to be devising ways to mine the data and make sense of it all, or at least in part—to turn data into knowledge, and knowledge into wisdom. The leading example is Google, which uses its MapReduce framework to process over

20 petabytes of data per day [8]. While a majority of Fortune 1000 companies are en-route to understanding Hadoop and adopting it in their technology stack, several start-ups have started already asking the important and inevitable question: "What's Next?". Hadoop for the first time has allowed us to analyze massive amounts of data without necessarily indulging in expensive proprietary hardware or software. However, adoption of Hadoop alone isn't necessarily helping businesses make smarter decisions or discover completely new facts. The power of scalable infrastructure needs to be supplemented with nifty data mining and machine learning tools, better visualization of results, and easier ways to track and analyze the findings over a period of time. Besides, there is the entire realm of real-time analytics, which is beyond the batch oriented nature of Hadoop. Pig, Sawzall, Microsoft's Dryad, and others functional languages are in development. They can be classified by terms like high throughput computing (HTC) or many-task computing (MTC), depending on the amount of data and the number of tasks involved in the computation [35]. Although these systems differ in design, the programming models they provide share similar objectives, namely hiding the hassle of parallel programming, fault tolerance and execution optimizations from the developer. Developers can typically continue to write sequential programs. The processing framework then takes care of distributing the program among the available nodes and executes each instance of the program on the appropriate fragment of data.

Conceptually, many of the Big Data analysis can be thought of as single program multiple data (SPMD) [7] algorithms or a collection thereof. These SPMDs can be implemented using different parallelization techniques such as threads, MPI, MapReduce, and mash-up or workflow technologies, yielding different performance and usability characteristics. Most techniques try to explore an "almost embarrassingly parallel" style of parallelism (e.g., analysis of independent events in particle physics, or independent documents for information retrieval). In this case, the parallel independent sets of data lead to independent "maps" (processing), which are followed by a reduction (e.g., to give histograms in particle physics, or aggregated queries in web searches). The excellent quality of service (QoS) and ease of programming provided by the MapReduce programming model has gained itself a lot of traction for this type of problem. However, the architectural and performance limitations of the current MapReduce architectures make their use questionable for many applications (e.g., machine learning algorithms need iterative closely coupled computations). More general workflow or dataflow paradigm (which is seen in Dryad and MapReduce extensions) is always valuable, and we explore such solutions in the following paragraphs.

3.1 Runtime Environments for Big Data

High level languages (i.e., for parallel programming) have been a holy grail for computer science research, but lately researchers made a lot of progress in the area of runtime environments. There is much similarity between parallel and distributed run times, with both supporting messaging with different properties (several such choices are presented in Fig. 1, for different hardware and software models). The hardware support of parallelism/concurrency varies from shared memory multicore, closely coupled clusters, and higher-latency (possibly lower bandwidth) distributed systems. The

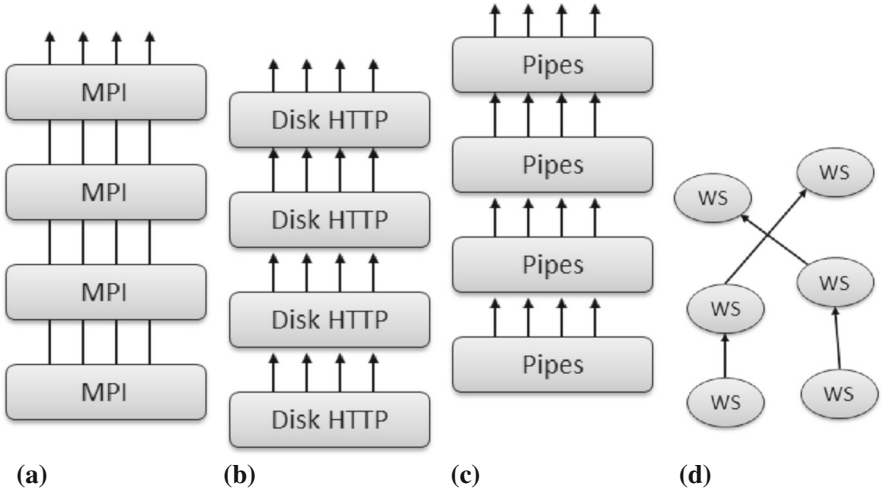

Fig. 1 Combinations of processes/threads and intercommunication mechanisms [12]. **a** MPI is long running processes with Rendezvoud for message exchange/synchronization. **b** Yahoo's Hadoop uses short running processes communicating via disk and tracking processes. **c** Microsoft's Dryad uses short running processes communicating via pipes, disk or shared memory between cores. **d** Web Services send irregular point-to-point messages between short or long running services

coordination (communication/synchronization) of the different execution units vary from threads (with shared memory on cores), MPI (between cores or nodes of a cluster), workflow or mash-ups linking services together, and the new generation of data intensive programming systems typified by Hadoop (implementing MapReduce) or Dryad.

Short running threads can be spawned up in the context of persistent data in memory and have modest overhead [12]. Short running processes (i.e., implemented as stateless services) are seen in Dryad and Hadoop. Also, various runtime platforms implement different patterns of operation. In *Iteration*-based platforms, the results of one stage are iterated many times. This is typical of most MPI style algorithms. In *Pipelining*-based platforms, the results of one stage (e.g., Map or Reduce operations) are forwarded to another. This is functional parallelism typical of workflow applications.

An important ambiguity in parallel/distributed programming models/runtimes comes from the fact that today both the parallel MPI style parallelism and the distributed Hadoop/Dryad/Web Service/Workflow models are implemented by messaging. This is motivated by the fact that messaging avoids errors seen in shared memory thread synchronization. MPI is a perfect example of runtimes crossing different application characteristics. MPI gives excellent performance and ease of programming for MapReduce, as it has elegant support for general reductions. However, it does not have the fault tolerance and flexibility of Hadoop or Dryad. Further MPI is designed for local computing; if the data is stored in a compute node's memory, that node's CPU is responsible for computing it. Hadoop and Dryad combine this idea with the notion of taking the computing to the data. A (non-comprehensive) presentation of technologies in use today for Big Data processing is presented in Fig. 2.

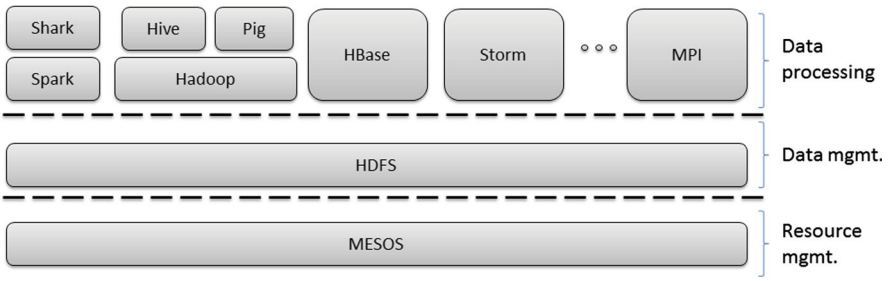

Fig. 2 Example of an ecosystem of Big Data analysis tools and frameworks

3.2 MapReduce and Hadoop

MapReduce (MR) emerged as an important programming model for large-scale data-parallel applications [8]. The MapReduce model popularized by Google is attractive for ad-hoc parallel processing of arbitrary data, and is today seen as an important programming model for large-scale data-parallel applications such as web indexing, data mining and scientific simulations, as it provides a simple model through which users can express relatively sophisticated distributed programs.

MapReduce breaks a computation into small tasks that run in parallel on multiple machines, and scales easily to very large clusters of inexpensive commodity computers. A MR program consists only of two functions, called Map and Reduce, written by a user to process key/value data pairs. The input data set is stored in a collection of partitions in a distributed file system deployed on each node in the cluster. The program is then injected into a distributed processing framework and executed in a manner to be described.

The Map function reads a set of "records" from an input file, does some filtering and/or transformations, and then outputs a set of intermediate records in the form of new key/value pairs. As the Map function produces these output records, a "split" function partitions the records into R disjoint buckets by applying a function to the key of each output record. This split function is typically a hash function, though any deterministic function will suffice. Each map bucket is written to the processing node's local disk. The Map function terminates having produced R output files, one for each bucket. In general, there are multiple instances of the Map function running on different nodes of a compute cluster. The term *instance* is used to refer to a unique running invocation of either the Map or Reduce function. Each Map instance is assigned a distinct portion of the input file by the MR scheduler to process. If there are M such distinct portions of the input file, then there are R files on disk storage for each of the M Map tasks, for a total of $M \times R$ files $F_{i,j}$, where $1 \leq i \leq M, 1 \leq j \leq R$. The key observation is that all Map instances use the same hash function; thus, all output records with the same hash value are stored in the same output file.

The second phase of a MR program executes R instances of the Reduce program (where R is typically the number of nodes). The input for each Reduce instance R_j consists of the files $F_{i,j}$, $1 \leq j \leq M$. These files are transferred over the network from the Map nodes' local disks. Again, all output records from the Map phase with the

same hash value are consumed by the same Reduce instance, regardless of which Map instance produced the data. Each Reduce instance processes or combines the records assigned to it in some way, and then writes records to an output file (in the distributed file system), which forms part of the computation's final output.

The input data set exists as a collection of one or more partitions in the distributed file system. It is the job of the MR scheduler to decide how many Map instances to run and how to allocate them to available nodes. Likewise, the scheduler must also decide on the number and location of nodes running Reduce instances. The MR central controller is responsible for coordinating the system activities on each node. A MR program finishes execution once the final result is written as new files in the distributed file system.

A key benefit of Map Reduce is that it automatically handles failures, hiding the complexity of fault-tolerance from the programmer. If a node crashes, MapReduce automatically reruns its tasks on a different machine. Similarly, if a node is available but is performing poorly, a condition called a *straggler*, MapReduce runs a *speculative copy* of its task (also called a "backup task") on another machine to finish the computation faster. Without this mechanism (known as "speculative execution"—not to be confused still with speculative execution at the OS or hardware level for branch prediction), a job would be as slow as the misbehaving task. In fact, Google has noted that in their implementation speculative execution can improve job response times by 44 % [8].

Google's MapReduce implementation is coupled with a distributed file system named Google file system (GFS) [15], from where it reads the data for MapReduce computations, and in the end stores the results. According to J. Dean et al., in their MapReduce implementation [8], the intermediate data are first written to the local files and then accessed by the reduce tasks. The same architecture is adopted by the Apache's MapReduce implementation, called *Hadoop*.

The popular open-source implementation of MapReduce, Hadoop [47], is developed primarily by Yahoo, where it runs jobs that produce hundreds of terabytes of data. Today Hadoop is used at Facebook, Amazon, etc. Researchers are using Hadoop for short tasks where low response time is critical: seismic simulations, natural language processing, mining web data, and many others. Hadoop includes several specific components.

First, Hadoop provides its own file system, *HDFS*. In HDFS, data is spread across the cluster (keeping multiple copies of it in case of hardware failures). The code is deployed in Hadoop to the machine that contains the data upon which it intends to operate on. HDFS organizes data by keys and values; each piece of data has a unique key and a value associated with that key. Relationships between keys can be defined only within the MapReduce application, not by HDFS.

On top of Hadoop, the developer defined a *MapReduce application*, as a functional programming paradigm that analysis a single record in HDFS, and then assembles the results into a consumable solution. The Mapper is responsible for the data processing step, while the Reducer receives the output from the Mappers and sorts the data that applies to the same key. A special process, called *Partitioner*, is responsible for dividing a particular analysis problem into workable chunks of data for use by the various Mappers. The *HashPartioner* is one example of a partitioner that is capable

to divide work up by "rows" of data in the HDFS. A developer can also create a *Combiner* for performing a local reduce that combines data before sending it back to Hadoop. The combiner performs the reduce step, which groups values together with their keys, but on a single node before returning the key/value pairs to Hadoop for proper reduction.

Additionally, Hadoop applications are deployed to an infrastructure that supports a high level of scalability and resilience. The infrastructure includes several specific components. The HDFS cluster is managed by a *NameNode*, that controls slave DataNode (i.e., nodes in charge of keeping the actual data) daemons. The NameNode manages locations of data, how the data is broken into blocks, what nodes those blocks are deployed to (and, generally, monitors the overall health of HDFS). Each cluster has one NameNode (and, unfortunately, in most current deployments the NameNode is a single-point of failure in a Hadoop cluster). An additional *Secondary NameNode* can also be used monitor the state of the HDFS cluster and take "snapshots" of the data contained in the NameNode. If the NameNode fails, then the Secondary NameNode can takes its place. This does require human intervention, however, so there is no automatic failover from the NameNode to the Secondary NameNode. Still, having the Secondary NameNode helps ensure that data loss is minimal.

Each slave node in the Hadoop cluster hosts a *DataNode*. The DataNode manages the data: it reads data blocks from the HDFS, manages the data on each physical node, and reports back to the NameNode with data management status. Also, a *JobTracker daemon* acts as liaison between the MapReduce application and Hadoop. There is one JobTracker configured per Hadoop cluster and, when one submits the code to be executed on the Hadoop cluster, it is the JobTracker's responsibility to build an execution plan. This execution plan includes determining the nodes that contain data to operate on, arranging nodes to correspond with data, monitoring running tasks, and relaunching tasks if they fail. Finally, similar to how data storage follows the master/slave architecture, all code execution follows the master/slave architecture. Each slave node has a *TaskTracker daemon* that is responsible for executing the tasks sent to it by the JobTracker and communicating the status of the job (and a heartbeat) with the JobTracker.

Hadoop's performance is closely tied to its *task scheduler*. In the original implementation the scheduler assumes that cluster nodes are homogeneous and tasks make progress linearly. Based on such assumptions, the scheduler is able to speculatively re-execute tasks that appear to be stragglers. Various studies show stragglers to appear for various reasons, including faulty hardware or misconfiguration.

As shown in [47], in practice the homogeneity assumptions do not always hold (e.g., in a virtualized data center such as Amazon's Elastic Compute Cloud, such assumptions lead to severe performance degradation). This is why authors proposed alternative scheduling algorithms for Hadoop, such as the *Longest Approximate Time to End*, which can show great performance advantages (especially in heterogeneous clusters).

At a higher level, there are several challenges related to application development on top of Hadoop. Users just analyzing data in a standalone system, with Hadoop clusters running in isolation, usually find it challenging to keep the clusters up and running. For others, challenges also relate to single points of failure, and massive

amounts of system that must work together, which makes system provisioning and management difficult. Other issues relate to the latency of execution in an application, or the inability to execute workloads outside the Hadoop paradigm.

Hadoop stores the intermediate results of the computations in local disks, where the computation tasks are executed, and it informs the appropriate workers to retrieve (pull) them for further processing. The same approach is adopted by the Disco, an open source MapReduce runtime developed using a functional programming language named Erlang [11]. Although this strategy of writing intermediate result to the file system makes the above runtimes robust, it introduces an additional step and a considerable communication overhead, which can be a limiting factor for some MapReduce computations. However, Hadoop, Disco and other similar runtimes focus mainly on computations that utilize a single map/reduce computational unit. Iterative MapReduce computations are not well supported.

Even common operations like database *Join* are tricky to implement in the MapReduce model. Moreover, it is necessary to embed MapReduce computations in a scripting language in order to execute programs that require more than one reduction or sorting stage. Each MapReduce instantiation is self-contained and no automatic optimizations take place across their boundaries. In addition, the lack of any type-system support or integration between the MapReduce stages requires programmers to explicitly keep track of objects passed between these stages, and may complicate long-term maintenance and re-use of software components.

MapIterativeReduce, is an alternative framework which extends the MapReduce programming model to better support reduce-intensive applications, while substantially improving its efficiency by eliminating the implicit barrier between the Map and the Reduce phase [39].Typically, MapReduce applications lack explicit support for reduction in distributed processing runtimes. To achieve this, programmers must implement an additional aggregator that collects the output data from all reduce jobs and combines them into a single result. For workloads with a large number of reducers and large data volumes, this approach can prove inefficient. MapIterativeReduce, is a framework for reduce-intensive computations running on Azure clouds. It leverages the VM local disks to exploit data locality and opt for a simpler and more efficient communication scheme for the coordination between entities.

Map-Reduce-Merge is a model that adds to Map-Reduce a Merge phase that can efficiently merge data already partitioned and sorted (or hashed) by map and reduce modules [44]. As indicated in [33], though sufficiently generic to perform many real world tasks, MapReduce is best at handling homogeneous datasets. However, joining multiple heterogeneous datasets does not quite fit into the MapReduce model (although it still can be done with extra MapReduce steps). For a search engine, data processing problems involve, for example, tasks which can best be modeled as joins. For example, a search engine usually stores crawled URLs with their contents in a crawler database, inverted indexes in an index database, click or execution logs in a variety of log databases, and URL linkages along with miscellaneous URL properties in a webgraph database. Such databases are gigantic and distributed over a large cluster of nodes. Moreover, their creation takes data from multiple sources: index database needs both crawler and webgraph databases, a webgraph database needs both a crawler and a previous version of the webgraph database. The Map-Reduce-Merge programming

model retains the MapReduce's many great features, while adding relational algebra to the list of database principles it upholds. It also contains several configurable components that enable many data-processing patterns. Most notably, it allows implementing many relational operators, particularly joins.

3.3 Pig and Hive

During the 1970s, the database research community engaged in a contentious debate whether a program to access data in a DBMS should be written either by 1) stating what one wants, rather than presenting an algorithm for how to get it (the Relational model), or 2) presenting an algorithm for data access (the Codasyl case). In the end, the former view prevailed and the last 30 years is a testament to the value of relational database systems. Programs in high-level languages, such as SQL, are easier to write, easier to modify, and easier for a new person to understand. Codasyl was criticized for being similarly to an assembly language for DBMS access. MapReduce-like programming is today seen somewhat analogous to Codasyl programming: the developer has to write algorithms in a low-level language in order to perform record-level manipulation. However, evidence from the MapReduce community suggests that there is widespread sharing of MapReduce code fragments to do common tasks, such as joining data sets. To alleviate the burden of having to re-implement repetitive tasks, the MapReduce community is migrating high-level languages on top of the current interface to move such functionality into the run time. Pig [28] and Hive [38] are two notable projects in this direction.

Such domain-specific languages, developed on top of the MapReduce model to hide some of the complexity from the programmer, today offer a limited hybridization of declarative and imperative programs and generalize SQL's stored-procedure model. Some whole-query optimizations are automatically applied by these systems across MapReduce computation boundaries. However, these approaches adopt simple custom type systems and prove limited support for iterative computations.

An alternative tool on top of Hadoop is being developed by Facebook. *Hive* lets analysts crunch data atop Hadoop using something very similar to the structured query language (SQL) that has been widely used since the 80s. It is based on concepts such as tables, columns and partitions, providing a high-level query tool for accessing data from their existing Hadoop warehouses [38]. The result is a data warehouse layer built on top of Hadoop that allows for querying and managing structured data using a familiar SQL-like query language, HiveQL, and optional custom MapReduce scripts that may be plugged into queries. Hive converts HiveQL transformations to a series of MapReduce jobs and HDFS operations and applies several optimizations during the compilation process.

The Hive data model is organized into tables, partitions and buckets. The tables are similar to RDBMS tables and each corresponds to an HDFS directory. Each table can be divided into partitions that correspond to sub-directories within an HDFS table directory and each partition can be further divided into buckets which are stored as files within the HDFS directories.

It is important to note that Hive was designed for scalability, extensibility, and batch job handling, not for low latency performance or real-time queries. Hive query

response times for even the smallest jobs can be of the order of several minutes and for larger jobs, may be on the order of several hours. Also, today Hive is the Facebook's primary tool for analyzing the performance of online ads, among other things.

Pig is a high-level data-flow language (Pig Latin) and execution framework whose compiler produces sequences of Map/Reduce programs for execution within Hadoop [28]. Pig is designed for batch processing of data. It offers SQL-style high-level data manipulation constructs, which can be assembled in an explicit dataflow and interleaved with custom Map- and Reduce-style functions or executables. Pig programs are compiled into sequences of Map-Reduce jobs, and executed in the Hadoop Map-Reduce environment.

The Pig data model contains scalar types which contain a single atomic value (integer, long, etc.), and three complex types which can contain other types: *Tuple* is a data record consisting of a sequence of 'fields', which can be any data type; *Bag* is a set of tuples, similar to a 'table'; *Map* is a map of a string key to a value, which can be any data type. Pig provides a set of operators for data processing. For example: LOAD and STORE can be used for reading and writing data from HDFS. Processing every tuple of a data set can use the FOREACH operator. Many operators are similar as SQL, such as JOIN, GROUP BY, UNION for standard data operations. As with many SQL implementations, Pig supports User-Defined Functions (UDF) which allows performing tasks written in low level language (Java or Python) to extend Pig.

Pig's infrastructure layer consists of a compiler that turns (relatively short) Pig Latin programs into sequences of MapReduce programs. Pig is a Java client-side application, and users install locally—nothing is altered on the Hadoop cluster itself. Grunt is the Pig interactive shell. With the support of this infrastructure, among the important advantages of Pig we mention the optimized data reading performance, the semi-structured data, and modular design. However, several limitations should not be ignored, such as the large amount of boiler-plate Java code (although proportionally less than Hadoop), the effort for learning how to use Pig and the lack of debugging techniques.

Still, such initiatives led to Hadoop being used today more and more extensively, from Twitter to eBay to LinkedIn. Facebook is only pushing the platform to new extremes. According to Jay Parikh [25], head of infrastructure at Facebook, today the company alone runs the world's largest Hadoop cluster—just one of several Hadoop clusters operated by Facebook spans more than 4,000 machines, and it houses over 100 petabytes of data, aka hundreds of millions of gigabytes.

3.4 Spark and Twister

MapReduce was highly successful in implementing large-scale data-intensive applications on commodity clusters. However, the model is built around an acyclic data flow model. This was soon followed by other models (or extensions to the MapReduce model), developed for example for classes of applications needing to reuse a working set of data across multiple parallel. This includes many iterative machine learning algorithms, as well as interactive data analysis tools. With such extensions, the MapReduce programming model can be applied also to fields such as data clustering, machine learning, and computer vision where many iterative algorithms are

common. In these algorithms, MapReduce is used to handle the parallelism while the repetitive application of it completes the iterations.

Spark is a framework that supports such applications while retaining the scalability and fault tolerance of MapReduce [46]. Spark provides two main abstractions for parallel programming: *resilient distributed datasets* and *parallel operations* on these datasets (invoked by passing a function to apply on a dataset).

Resilient distributed datasets (RDDs) are read-only collections of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Users can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like parallel operations zaharia2012resilient. RDDs achieve fault tolerance through a notion of lineage: if a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to be able to rebuild just that partition.

To use Spark, developers write a *driver program* that implements the high-level control flow of their application and is responsible for launching operations in parallel. Programmers then invoke operations like *map*, *filter* and *reduce* by passing closures (functions) to Spark. These closures can further refer to variables in the scope where they are created. Normally, when Spark runs a closure on a worker node, these variables are copied to the worker. However, Spark also lets programmers create restricted types of shared variables. Spark supports several parallel operations. A *reduce* operation can be used to combine dataset elements using an associative function to produce a result at the driver program. The *collect* operation sends all elements of the dataset to the driver program. For example, an easy way to update an array in parallel is to parallelize, map and collect the array. The *foreach* operation passes each element through a user provided function. This is only done for the side effects of the function (which might be to copy data to another system or to update a shared variable). In addition, Spark supports two restricted types of shared variables that can be used in functions running on the cluster.

Spark is implemented in Scala, a statically typed high-level programming language for the Java VM, and exposes a functional programming interface similar to DryadLINQ. Spark can also be used interactively, and allows the user to define RDDs, functions, variables and classes and use them in parallel operations on a cluster. According to experiments presented by authors of [46], by making use extensively of memory storage (using the RDD abstractions) of cluster nodes, most of the operations Spark can outperform Hadoop by a factor of ten in iterative machine learning jobs, and can be used to interactively query a large dataset with sub-second response time.

Twister is another MapReduce extension, designed to support iterative MapReduce computations efficiently [10]. Twister uses a publish/subscribe messaging infrastructure for communication and data transfers, and supports long running map/reduce tasks, which can be used in "configure once and use many times" approach. In addition, it provides programming extensions to MapReduce with "broadcast" and "scatter" type data transfers. It also allows long-lived map tasks to keep static data in memory between jobs in a manner of "configure once, and run many times".

Such improvements allow Twister to support iterative MapReduce computations highly efficiently compared to other MapReduce runtimes [10]. To achieve this, Twister handles the intermediate data in the distributed memory of the worker nodes.

The results of the *map* tasks are directly pushed via the broker network to the appropriate *reduce* tasks where they get buffered until the execution of the *reduce* computation. Therefore, Twister assumes that the intermediate data produced after the map stage of the computation will fit in to the distributed memory. To support scenarios with large intermediate results, one can extend the Twister runtime to store the reduce inputs in local disks instead of buffering in memory. But Hadoop is not the only Big Data option. Hadoop is, in fact, a tool with a lot of potential for solving many Big Data problems, but it might not be the best tool for every Big Data situation, as we will see next.

3.5 Dryad and DryadLINQ

Dryad is a general-purpose distributed execution engine for coarse-grain data-parallel applications [21]. While MapReduce was designed to be accessible to the widest possible class of developers (aiming for simplicity at the expense of generality and performance), the Dryad system allows the developer fine control over the communication graph as well as the subroutines that live at its vertices. A Dryad application developer can specify an arbitrary directed acyclic graph to describe the application's communication patterns, and express the data transport mechanisms (files, TCP pipes, and shared-memory FIFOs) between the computation vertices.

A Dryad application combines computational "vertices" with communication "channels" to form a dataflow graph. In other words, a Dryad job is similar to a directed acyclic graph where each vertex is a program and edges represent data channels. At run time, vertices are processes communicating with each other through the channels, and each channel is used to transport a finite sequence of data records.

Dryad runs the application by executing the vertices of this graph on a set of available computers, communicating as appropriate through files, TCP pipes, and shared-memory FIFOs. The vertices provided by the application developer are quite simple and are usually written as sequential programs with no thread creation or locking. Concurrency arises from Dryad scheduling vertices to run simultaneously on multiple computers, or on multiple CPU cores within a computer. The application can discover the size and placement of data at run time, and modify the graph as the computation progresses to make efficient use of the available resources.

Dryad is designed to scale from powerful multi-core single computers, through small clusters of computers, to data centers with thousands of computers. The Dryad execution engine handles all the difficult problems of creating a large distributed, concurrent application: scheduling the use of computers and their CPUs, recovering from communication or computer failures, and transporting data between vertices.

The execution of a Dryad job is orchestrated by a centralized *job manager*. The job manager is responsible for instantiating a job's dataflow graph, scheduling processes on cluster computers, providing fault-tolerance by re-executing failed or slow processes, monitoring the job and collecting statistics, and transforming the job graph dynamically according to user-supplied policies. It contains the application-specific code to construct the job's communication graph along with library code to schedule the work across the available resources. All data is sent directly between vertices and thus the job manager is only responsible for control decisions and is not a bottleneck for any data transfers.

A cluster is typically controlled by a task scheduler, separate from Dryad, which manages a batch queue of jobs and executes a few at a time subject to cluster policy. Microsoft [21] uses in its implementation a distributed storage system that shares with the Google file system the property that large files can be broken into small pieces that are replicated and distributed across the local disks of the cluster computers. But Dryad also supports the use of NTFS for accessing files directly on local computers, which can be convenient for small clusters with low management overhead.

Nephele is another alternative data processing framework that allows assigning the particular tasks of a processing job to different types of virtual machines and takes care of their instantiation and termination during the job execution [43]. Unlike Dryad, who runs jobs described as DAGs and offers the possibility to connect the involved tasks through either file, network or in-memory channels, Nephele exploits the dynamic resource provisioning offered by today's compute clouds. Dryad assumes an execution environment which consists of a fixed set of homogeneous worker nodes. Dryad scheduler is designed to distribute tasks across the available compute nodes in a way that optimizes the throughput of the overall cluster. It does not include the notion of processing cost for particular jobs. Unlike this, Nephele allows each task to be executed on its own instance type, so the characteristics of a requested virtual machine can be adapted to the demands of each processing phase.

DryadLINQ is a system and a set of language extensions that enable a programming model for large scale distributed computing [45]. It generalizes execution environments such as SQL, MapReduce, and Dryad in two ways: by adopting an expressive data model of strongly typed .NET objects; and by supporting general-purpose imperative and declarative operations on datasets within a traditional high-level programming language. A DryadLINQ application is a sequential program (hence, the programmer is given the "illusion" of writing for a single computer), composed of LINQ (Language Integrated Query) expressions performing imperative or declarative operations and transformations on datasets, and can be written and debugged using standard .NET development tools. Objects in DryadLINQ datasets can be of any .NET type, making it easy to compute with data such as image patches, vectors, and matrices. DryadLINQ programs can use traditional structuring constructs such as functions, modules, and libraries, and express iteration using standard loops. Crucially, the distributed execution layer employs a fully functional, declarative description of the data-parallel component of the computation, which enables sophisticated rewritings and optimizations like those traditionally employed by parallel databases. The DryadLINQ system automatically and transparently translates the data-parallel portions of the program into a distributed execution plan which is passed to the Dryad execution platform, which further ensures efficient, reliable execution of this plan.

3.6 Piccolo

As presented, MapReduce and Dryad provide a data-flow programming model suited for bulk-processing of on-disk data. However, they are not necessarily a natural fit for in-memory computation: they do not expose any globally shared state, and applications have no online access to intermediate state and often have to emulate shared memory

access by joining multiple data streams. *Piccolo* is a data-centric programming model for writing parallel in-memory applications across many machines [34].

Applications written in Picollo consist of *control* functions, which are executed on a single machine, and *kernel* functions, which are executed concurrently on many machines. Thus, programmers organize their computation around a series of application kernel functions, where each kernel is launched as multiple instances concurrently executing on many compute nodes. Control functions create shared tables, launch multiple instances of a kernel function, and perform global synchronization. Kernel functions are sequential code which read from and write to tables to share state among concurrently executing kernel instances. Thus, kernel instances share distributed, mutable state using a set of in-memory tables whose entries reside in the memory of different compute nodes. Kernel instances share state exclusively via the key-value table interface with get and put primitives. The underlying Piccolo runtime sends messages to read and modify table entries stored in the memory of remote nodes.

By exposing shared global state, the programming model of Piccolo offers several attractive features. First, it allows for natural and efficient implementations for applications that require sharing of intermediate state (such as k-means computation, n-body simulation, PageRank calculation etc.). Second, Piccolo enables online applications that require immediate access to modified shared state. For example, a distributed crawler can learn of newly discovered pages quickly as a result of state updates done by ongoing web crawls.

Piccolo borrows ideas from existing data-centric systems to enable efficient application implementations. Piccolo enforces atomic operations on individual key-value pairs and uses user-defined accumulation functions to automatically combine concurrent updates on the same key (this is somewhat similar to the reduce operation in MapReduce). The combination of these two techniques eliminates the need for fine-grained application-level synchronization for most applications. Piccolo allows applications to exploit locality of access to shared state. Users control how table entries are partitioned across machines by defining a partitioning function. Based on users' locality policies, the underlying run-time can schedule a kernel instance where its needed table partitions are stored, thereby reducing expensive remote table access. Piccolo run-time system actually consists of one master (used for coordination) and several worker processes (in charge of storing in-memory table partitions and executing kernels). The run-time uses a simple work stealing heuristic to dynamically balance the load of kernel execution among workers. Piccolo provides a global checkpoint/restore mechanism to recover from machine failures. For this, the run-time uses the Chandy-Lamport snapshot algorithm to periodically generate a consistent snapshot of the execution state without pausing active computations. Upon machine failure, Piccolo recovers by restarting the computation from its latest snapshot state.

Experiments have shown that Piccolo is fast and provides excellent scaling for many applications. The performance of PageRank and k-means on Piccolo is 11x and 4x faster than that of Hadoop [34]. Or, as another example, computing a PageRank iteration for a 1 billion-page web graph takes 70 s on 100 EC2 instances.

3.7 Programming Languages

Although MapReduce and similar programming models prove quite effective, developers need adequate on-top supporting languages to develop their applications. Attaching existing languages such as Python to MapReduce is insufficient for several reasons. Notation customized to a particular problem domain make programs clearer, more compact, and more expressive. Support for protocol buffers and other domain-specific types simplifies programming at a lower level. And, advantages of a custom language include the ability to add domain-specific features, custom debugging and profiling interfaces, and so on.

Sawzall is a procedural domain-specific programming language used by Google to process large numbers of individual log records [33]. Sawzall was first described in 2003, but the runtime was open-sourced in August 2010.

The motivation behind the launch of Sawzall relates to how Google's server logs are stored as large collections of records (protocol buffers), partitioned over many disks within GFS. In order to perform calculations involving the logs, MapReduce programs can be written in C++ or Java, which engineers soon discovered to be too time-consuming pike2005interpreting. To make it easier to write quick scripts, Rob Pike et al. developed the Sawzall language. A Sawzall script runs within the Map phase of a MapReduce and "emits" values to tables. Then the Reduce phase (which the script writer does not have to be concerned about) aggregates the tables from multiple runs into a single set of tables.

The most important motivation for Sawzall's creation relates to parallelism. Capturing the aggregators in the language (and its environment) means that the programmer never has to provide one, unlike when using traditional MapReduce programs. This also leads to more elegant programs, as well as a comfortable way to think about data processing problems in large distributed data sets. Separating out the aggregators and providing no other inter-record analysis maximizes the opportunity to distribute processing across records. It also provides a model for distributed processing, which in turn encourages users to think about the problem in a different light. In a traditional language such as Awk or Python, users would be tempted to write the aggregators in that language, which would be difficult to parallelize. Even if one provided a clean interface and library for aggregators in these languages, seasoned users would want to roll their own sometimes, which could introduce dramatic performance problems.

The model that Sawzall provides has proven valuable. Although some problems, such as database joins, are poor fits to the model, most of the processing done on Google on large data sets fits well and the benefit in notation, convenience. Also, one unexpected benefit of the system arose from the constraints the programming model places on the user. Since the data flow from the input records to the Sawzall program is so well structured, it was easy to adapt it to provide fine-grained access control to individual fields within records. The system can automatically and securely wrap the user's program with a layer, itself implemented in Sawzall, that elides any sensitive fields. For instance, production engineers can be granted access to performance and monitoring information without exposing any traffic data.

A different approach to the processing of large data stores is to analyze them with a data stream model. Such systems process the data as it flows in, and their

operators are dependent on the order of the input records. For example, *Aurora* [1] is a stream processing system that supports a (potentially large) set of standing queries on streams of data. Analogous to Sawzall's predefinition of its aggregators, Aurora provides a small, fixed set of operators, although two of them are escapes to user-defined functions. These operators can be composed to create more interesting queries. Unlike Sawzall, some Aurora operators work on a contiguous sequence, or window, of input values. Aurora only keeps a limited amount of data on hand, and is not designed for querying large archival stores. There is a facility to add new queries to the system, but they only operate on the recent past. Aurora's efficiency comes from a carefully designed run-time system and a query optimizer, rather than Sawzall's brute force parallel style.

Another stream processing system, *Hancock* [6], goes further and provides extensive support for storing per-query intermediate state. This is quite a contrast to Sawzall, which deliberately reverts to its initialization state after each input record. Like Aurora, Hancock concentrates on efficient operation of a single thread instead of massive parallelism.

It may seem paradoxical to use an interpreted language in a high-throughput environment, but reality shows that the CPU time is rarely the limiting factor; the expressibility of the language means that most programs are small and spend most of their time in I/O and native run-time code [33]. Moreover, the flexibility of an interpreted implementation has been helpful, both in ease of experimentation at the linguistic level and in allowing us to exploreways to distribute the calculation across many machines. Overall, for example, Sawzall programs tend to be around 10–20 times shorter than the equivalent MapReduce programs in C++ and significantly easier to write [33].

3.8 Sharing the Data and Online Processing

With all the BigData tools and instruments available, we are still far from understanding all the complexities behind processing large amounts of data. For example, the data analysis was previously regarded with respect to mainly one and only objective—scientists working on human genome use their data for this objective delisi2008meetings. While such examples represent silo approaches to Big Data applications, they can also undermine potential opportunities. For example, scientists working on human genome data may improve their analysis if they could take all publications on Medline and analyze it in conjunction with the human genome data. However, this requires natural language processing (semantic) technology combined with bioinformatics algorithms, an unusual coupling at best. The question is how to best support this "sharing" of Big Data?

Recent projects such as *BigQuery* have the potential to encourage scientists to put their data into the Cloud, where potentially others might have access as well [20]. BigQuery is a tool developed by Google to allow ordinary users run ad hoc queries using an SQL-like syntax. Google had used previously the tool (under the name Dremel) internally for years before releasing a form of it in their generally available service - BigQuery—capable to get results in seconds from terabytes of data [40]. The tool is hosted on Google's infrastructure. It's main advantage is simplicity:

compared to Hadoop, which requires set up and administration, companies can take their data, put it in Google's cloud, and use it directly into their applications.

BigQuery is a ready-to-use tool, fast to set up and capable to bring results fast, even from very big datasets. The user simply takes his data, put it in Google's Cloud, and use the application. Unlike other tools, BigQuery is capable to return results in seconds from terabytes of data. Google has used the tool (in a version called Dremel) internally for years before releasing it to the public in 2012. Today many companies use BigQuery. When the data becomes too big, tools such as Hadoop might not be sufficient for small companies, because they require set up and administration. Small companies do not have time to set up hardware, they just need to run queries on data and integrate it into their application, and this is where BigQuery fits very well [19].

Yahoo!'s S4 (Simple Scalable Streaming System) [26] is a general-purpose, distributed, scalable, partially fault-tolerant, pluggable platform that allows programmers to easily develop applications for processing continuous unbounded streams of data. Its architecture provides semantics of encapsulation and location transparency, thus allowing applications to be massively concurrent while exposing a simple programming interface to application developers. Its design is primarily driven by large scale applications for data mining and machine learning in a production environment.

Another example is Microsoft's Windows Azure [22]—its cloud operating system. It is based on services such as Live Services (Live Mesh), SQL Services, SharePoint Services, .NET Services and/or Dynamics CRM Services, run from Data Centers spread over the entire globe and linked together by individual servers. Each such individual server spreads its memory and network load with others. Azure offers several components. Compute is the component that runs applications in the cloud. These applications largely see a Windows Server environment. The Storage element stores binary and structured data in the cloud. The Fabric Controller deploys, manages, and monitors applications. The fabric controller also handles updates to system software throughout the platform. All these components are united by the Content Delivery Network (CDN), which speeds up global access to binary data in Windows Azure storage by maintaining cached copies of that data around the world. Finally, Connect allows creating IP-level connections between on-premises computers and Windows Azure applications. Just to present an example, applications normally created on Windows can be parallelized by allowing a number of Workers run simultaneously within Microsoft's Data Center, and take up the parallel work.

ConPaaS is an integrated Cloud platform, developed in the framework of the EU FP7 Contrail project, for Big Data [32]. It allows developers easily write scalable Cloud applications without worrying about the complexity of the Cloud. The platform provides a runtime environment that facilitates deployment of end-user applications in the Cloud. In ConPaaS, applications are organized as a collection of services. Using these services a bioinformatics application could, for example, be composed of a MapReduce service backend to process genomic data, as well as a Web hosting and SQL database service to provide a Web-based graphical interface to the users. Each service can be scaled on demand to adjust the quantity of computing resources to the capacity needs of the application.

ConPaaS contains two services specifically dedicated to Big Data: MapReduce and TaskFarming. MapReduce provides users with the well-known parallel programming

paradigm. TaskFarming allows the automatic execution of a large collection of independent tasks such as those issued by Monte-Carlo simulations. The ability of these services to dynamically vary the number of Cloud resources they use makes it well-suited to very large computations: one only needs to scale services up before a big computation, and scale them down afterwards.

An important element in all Big Data applications is the requirement for a scalable file system where input and output data can be efficiently stored and retrieved. ConPaaS comes together with the XtreemFS distributed file system for clouds. Like ConPaaS services, XtreemFS is designed to be highly available and fully scalable. Unlike most other file systems for the Cloud, XtreemFS provides a POSIX API. This means that an XtreemFS volume can be mounted locally, giving transparent access to files in the Cloud.

Similarly, Facebook is today building Prism [25] a platform currently rolling out across the Facebook infrastructure. The typical Hadoop cluster is governed by a single "namespace" and a list of computing resources available for each job. In opposition, Prism carves out multiple namespaces, creating many "logical clusters" that operate atop the same physical cluster. Such names spaces can then be divided across various Facebook teams, and all of them would still have access to a common dataset that can span multiple data centers. *Nexus* is a low-level substrate that provides isolation and efficient resource sharing across frameworks running on the same cluster, while giving each framework freedom to implement its own programming model and fully control the execution of its jobs [18]. As new programming models and new frameworks emerge, they will need to share computing resources and data sets. For example, a company using Hadoop should not have to build a second cluster and copy data into it to run a Dryad job. Sharing resources between frameworks is difficult today because frameworks perform both job execution management and resource management. For example, Hadoop acts like a "cluster OS" that allocates resources among users in addition to running jobs. To enable diverse frameworks to coexist, Nexus decouples job execution management from resource management by providing a simple resource management layer over which frameworks like Hadoop and Dryad can run.

Nexus provides a "slot" abstraction, in which frameworks may run "tasks" that do arbitrary work. Along with a mechanism called "slot offers", the fine granularity of slots lets Nexus achieve more efficient resource sharing across frameworks than would be possible with traditional coarse-grained cluster scheduling systems. Nexus is analogous to a "cluster hypervisor": it provides isolation and multiplexing while giving each framework a high level of control over its own execution.

Mesos is a thin resource sharing layer that enables fine-grained sharing across diverse cluster computing frameworks, by giving frameworks such as Hadoop or Dryad a common interface for accessing cluster resources [17]. To support a scalable and efficient sharing system for a wide array of processing frameworks, Mesos delegates control over scheduling to the framework themselves. This is accomplished through an abstraction called a *resource offer*, which encapsulates a bundle of resources that a framework can allocate on a cluster node to run tasks. Mesos decides how many resources to offer each framework, based on an organizational policy such as fair sharing, while frameworks decide which resources to accept and which tasks to run on them. While this decentralized scheduling model may not always lead to globally

optimal scheduling, in practice its developers found that it performs surprisingly well in practice, allowing frameworks to meet goals such as data locality nearly perfectly [17]. In addition, resource offers are simple and efficient to implement, allowing Mesos to be highly scalable and robust to failures.

Solutions such as these already uncover a whole new generation of BigData "super software" tools. Today, in the department of platforms, Apache Hadoop's open source software enables the distributed processing of large data sets across clusters of commodity servers. Other similar cluster computing frameworks continue to emerge, and today it seems clear that no such framework will probably be optimal for all applications. Therefore, organizations will probably run multiple frameworks in the same cluster, using the best one for each application. Multiplexing a cluster between frameworks can, indeed, improve utilization and allow applications to share access to large datasets that may be otherwise too costly to replicate across clusters. In this sense, IBM's Platform Symphony is an example of cloud management solution suitable for a variety of distributed computing and Big Data analytics applications. Oracle, HP, SAP, and Software AG are very much in the game for this $10 billion industry. While these giants are offering variety of solutions for distributed computing platforms, there is still a huge void at the level of Analytics Super Software.

So, what would a Super Software be doing at the top of the pyramid? This component would be managing multiple applications under its umbrella, almost like an auto-pilot for airplanes, Super Software's main function would be to discover new knowledge which would otherwise be impossible to acquire via manual means. To that end, discovery would require functions such as: finding associations across information in any format; visualization of associations; search; categorization, compacting, summarization; characterization of new data (where it fits); alerting; and, finally, cleaning (deleting unnecessary clogging information).

In genetics, such a Super Software would be able to identify genetic patterns of a disease from human genome data, supported by clinical results reported in Medline, and further analyzed to unveil mutation possibilities using FBI's DNA bank of millions of DNA information. One can extend the scope and meaning of top level objectives which is only limited by our imagination. With Big Data, we have finally reached a cross-road where computers are going to "have" to think on behalf of humans to discover new knowledge [37]. There is no turning back anymore, we have reached the limit of mental capacity to cope, let alone absorb information to process into knowledge.

4 Challenges

Sciences and industry are currently undergoing a profound transformation: large-scale, diverse data sets and streams (derived from sensors, the web, transactions, or complex simulations) present a huge opportunity for data-driven decision making.

Besides the *sheer volume of data*, "Big Data" will come in a *variety of data formats* (e.g., sensor data, text, audio, video), origin, quality, and so forth. The data are *created at an ever-increasing rate*, making the subject of velocity (i.e., the time window in which the data will need to be processed) crucial in order to arrive at actionable information in a timely manner. Moreover, as the data come from different sources

of different quality and trustworthiness, another crucial aspect of data analytics is to *assess the veracity of an analysis result*, i.e., its correctness and credibility. Furthermore, the *visualization and interactive analysis* of huge, changing data sets still presents numerous challenges for data analysts.

Recent advanced in computer technology and processing paradigms have created new tools and technologies at the forefront of "Big Data". On top of such tools, the "four V's" (Volume, Velocity, Variety, and Veracity) put pressure on developers to become comfortable with new programming paradigms. And the domain is continuously attracting significant attention in society, industry, and science. Novel statistical and mathematical algorithms, prediction techniques, and modeling methods, new approaches for data collection and integration, data analysis and compression, enhanced technologies for processing and sharing data and information, as well as novel languages for the declarative specification and automatic optimization and parallelization of complex data analysis programs are needed to *simultaneously cope with the volume, velocity, variety, and veracity aspects* of data analytics. This capability will enable a paradigm shift in scientific and commercial applications. Advances in information processing, integration, signal processing, machine learning, data mining, compression, and visualization will open up new ways of extracting useful, reliable, and verified information in a timely fashion from huge and diverse data sets.

The "NoSQL" approach brings with it a range of issues. *Integrating Big Data of various media types* and *providing low latency and high velocity analytics with trustworthy information* is a major challenge not met by existing data management systems. Big Data analytics systems must be able to ingest data from various media types, in particular audio and video streams, at increasing speeds, while at the same time already enabling the analysis of the data in a continuous fashion. Scalable, easy-to-use database or data analysis systems and new algorithms and analysis paradigms must be developed that address such different aspects and requirements simultaneously. Some examples are scalable online analysis, computational linguistics, statistics, and machine learning algorithms. The specification and automatic optimization of data analysis programs that inclu3de iterations and complex user-defined functions in a scalable way for a variety of hardware platforms such as SIMD, clusters, or many-core CPUs, as well as complex hardware architectures, such as NUMA still presents many research challenges. Other examples of challenges are new declarative languages and methods for the scalable processing and optimization of complex data analysis programs, such as active learning techniques or interactive entity linking techniques as part of computational linguistics, which must consider latency as a hard constraint, while ensuring a certain degree of trustworthiness in the case of contradicting, missing, or incomplete information, even in resource-constrained environments.

The *recent data explosion* is going to make life difficult in many industries, and those companies that can adapt well and gain the ability to analyze such data will have a considerable advantage over those that lag. *New skill sets* are going to be needed, and these skills will be scarce. Programming for Big Data applications is an altogether trickier affair, and IT departments that are staffed with people who understand SQL are ill-equipped to tackle the world of MapReduce programming, parallel programming and key-value databases that is starting to represent the state of the art in tackling very large data sets. The old generation of programmers and software products that relied

pretty-much on a common standard for database access will need to adapt to a world where understanding internal database structure will allow considerable productivity gains.

Companies *need to explore the newer approaches to handling large data volumes and begin to understand the limitations and challenges that come with technologies* like Hadoop and NoSQL databases, if they are to avoid being swept away by the Big Data tidal wave. Certainly, there is currently considerable enthusiasm around the MapReduce paradigm for large scale data analysis [30]. But the basic control flow of this framework has existed in parallel SQL database management systems (DBMS) for over 20 years. Parallel database systems (which all share a common architectural design) have been commercially available for nearly two decades, with many still existing in the marketplace, including Teradata, Aster Data, Netezza, DATAllegro (and therefore soon Microsoft SQL Server via Project Madison), Dataupia, Vertica, ParAccel, Neoview, Greenplum, DB2 (via the Database Partitioning Feature), and Oracle (via Exadata), to name a few. They are robust, high performance computing platforms. Similar to MapReduce, they all provide a high-level programming environment and parallelize readily. Though it may seem that MR and parallel databases target different audiences, it is in fact possible to write almost any parallel processing task as either a set of database queries (possibly using user defined functions and aggregates to filter and combine data) or a set of MapReduce jobs. Thus, evidence is needed to understand the differences between the MapReduce approach to performing large-scale data analysis and the approach taken by parallel database systems. The two classes of systems make different choices in several key areas. For example, all DBMSs require that data conform to a well-defined schema, whereas MR permits data to be in any arbitrary format. Other differences include how each system provides indexing and compression optimizations, programming models, the way in which data is distributed, and query execution strategies.

Authors of [31] present interesting results for benchmarks executed on a 100-node cluster with Hadoop versus parallel SQL DBMSs, such as Vertica: SQL DBMSs can run significantly faster and the required less code to implement each task, but take longer to tune and load the data. Such differences can be explained in various ways: Despite the increased complexity of the query, *the performance of Hadoop is limited by the speed with which its tables can be read off disk*. A MR program occasionally has to perform complete table scans, while parallel database systems are able to take advantage of clustered indexes to reduce the amount of data that needed to be read. Also, the *parallel DBMSs are able to take advantage of partitioned tables*—such systems are able to do a join locally on each node, without any network overhead of repartitioning before the join. In our opinion there is a lot to learn from both kinds of systems. Most importantly is that higher level interfaces, such as Pig [28] and Hive [38], are continuously being placed on top of the MR foundation, and a number of tools similar more expressive than MR, such as Dryad [21], are needed. This will make complex tasks easier to code in MR-style systems and remove one of the big advantages of SQL engines, namely that they take much less code on the tasks in our benchmark. For parallel databases, we believe that both commercial and open-source systems will dramatically improve the parallelization of user-defined functions. Hence, the APIs of the two classes of systems are clearly moving toward each other. Early evidence of

this is seen in the solutions for integrating SQL with MR offered by Greenplum and Asterdata [41].

Another challenge is generated by the *improvement of data quality for Big Data*, as researchers and practitioners aim to evaluate the "fitness for use" of data sets. Most of the proposed methods and techniques are based on two main assumptions: data are structured, and the purposes for which data are used are known. Clearly, these two assumptions are not valid in the Big Data environment in which data are integrated from heterogeneous sources and may not be generated by well-defined processes—as in general they are used to satisfy unanticipated and different requirements. In such scenario, new assessment techniques have to be proposed and additional data quality dimensions have to be defined. In particular, some efforts should be directed toward the evaluation of the value of the data relative both to intrinsic dimensions, such as accuracy, completeness, currency, and to dimensions related to data provenance, such as the credibility and reputation of the data sources.

A last challenge relates to the *applicability of programming model* such as the ones presented here to a large set of computational problems. The generality of such models was lately discussed by a number of authors [9,23]. Authors of [9], for example, argue that in many HPC applications, I/O is concurrently performed by all processes, which leads to I/O bursts. This causes resource contention and substantial variability of I/O performance, which significantly impacts the overall application performance and, most importantly, its predictability over time. For example, a typical behavior in large-scale simulations consists of alternating computation phases and write phases. As a rule of thumb, it is commonly accepted that a simulation spends at most 5 % of its run time in I/O phases. Often due to explicit barriers or communication phases, all processes perform I/O at the same time, causing network and file system contention. To address this issue, developers usually elaborate algorithms at the MPI-IO level to maintain a high throughput (but optimizations usually rely on all-to-all communications that impact their scalability). Damaris, which is the framework proposed by the authors, solve specifically such problems in case of large-scale simulations. The performance improvements compared to "all-generic" programming approaches for such applications (authors compared with a standard MPI implementation) leads us to think that obtained relatively good performance for applications in the future will also involve coping with specific requirements for particular classes of application. In other words, parallel and distributed programming models developed a lot, but they prove real value only when the developer truly understands how to specify applications correctly in terms of parallel constructs and using the right tool for the right job. Such solutions clearly show that abstractions and languages that are currently proposed to manage Big Data are heavily influenced by the underlying ICT platforms; but they are unsuitable for supporting "computational interdisciplinarity", as it is required if one wants to use the best of, e.g., analytical, inductive, and simulation techniques, all at work on the same data. Many platforms are dedicated to vertical application domains. Due to such limitations, most data computation scientists specialize in using given methods and become tied to the options and limitations of that method. In other words, *our society is data-rich, but it lacks the conceptual tools to handle it*.

5 Conclusions

With Cloud Computing emerging as a promising new approach for ad-hoc parallel data processing, major companies have started to integrate frameworks for parallel data processing in their product portfolio, making it easy for customers to access these services and to deploy their programs. We have entered the Era of Big Data. There are various mechanisms that generate Big Data: data from scientific measurements and experiments (astronomy, physics, genetics, etc.), peer-to-peer communication (text messaging, chat lines, digital phone calls), broadcasting (News, blogs), social networking (Facebook, Twitter), authorship (digital books, magazines, Web pages, images, videos), administrative data (enterprise or government documents, legal and financial records), business data (e-commerce, stock markets, business intelligence, marketing, advertising). Such services generate clickstream data from millions of users every day, which is a potential gold mine for understanding access patterns and increasing ad revenue.

The explosion and profusion of available data in a wide range of application domains rise up new challenges and opportunities in a plethora of disciplines—ranging from science and engineering to biology and business. One major challenge is how to take advantage of the unprecedented scale of data—typically of heterogeneous nature—in order to acquire further insights and knowledge for improving the quality of the offered services. To exploit this new resource, we need to scale up and scale out both our infrastructures and standard techniques. In this paper we analyzed opportunities and challenges for efficient parallel data processing. Big Data is the next frontier for innovation, competition, and productivity, and many solutions continue to appear, partly supported by the considerable enthusiasm around the MapReduce (MR) paradigm for large-scale data analysis. We reviewed various parallel and distributed programing paradigms, analyzing how they fit into the Big Data era, present modern emerging paradigms and frameworks. To better support practitioners interesting in this domain, we presented an analysis of on-going research challenges towards the truly fourth generation data-intensive science.

References

1. Abadi, D.J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: a new model and architecture for data stream management. VLDB J. Int. J. Very Large Data Bases **12**(2), 120–139 (2003)
2. Beckhusen, R.: So it begins: Darpa sets out to make computers that can teach themselves. http://www.wired.com/dangerroom/2013/03/darpa-machine-learning-2/all/1 (2013). Accessed 18 Apr 2013
3. Bell, G., Hey, T., Szalay, A.: Beyond the data deluge. Science **323**(5919), 1297–1298 (2009)
4. Berkan, R.: Big Data: a blessing and a curse. http://www.searchenginejournal.com/big-data-blessing/53528/ (2012). Accessed 15 Apr 2013
5. Cisco: Cisco visual networking index: Global mobile data traffic forecast update, 2011–2016. http://www.cisco.com/ (2012). Accessed 16 Apr 2013
6. Cortes, C., Fisher, K., Pregibon, D., Rogers, A.: Hancock: a language for extracting signatures from data streams. In: Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 9–17. ACM (2000)
7. Darema, F.: The spmd model: past, present and future. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface, pp. 1–1. Springer, Berlin (2001)

8. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
9. Dorier, M., Antoniu, G., Cappello, F., Snir, M., Orf, L.: Damaris: how to efficiently leverage multicore parallelism to achieve scalable, jitter-free i/o. In: 2012 IEEE International Conference on Cluster Computing (CLUSTER), pp. 155–163. IEEE (2012)
10. Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.H., Qiu, J., Fox, G.: Twister: a runtime for iterative mapreduce. In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, pp. 810–818. ACM (2010)
11. Ekanayake, J., Pallickara, S., Fox, G.: Mapreduce for data intensive scientific analyses. In: IEEE Fourth International Conference on eScience 2008 (eScience'08), pp. 277–284. IEEE (2008)
12. Fox, G., Bae, S.H., Ekanayake, J., Qiu, X., Yuan, H.: Parallel data mining from multicore to cloudy grids. In: High Performance Computing Workshop, vol. 18, pp. 311–340 (2009)
13. Frank, C.: Forbes: Improving Decision Making in the World of Big Data. http://www.forbes.com/sites/christopherfrank/2012/03/25/improving-decision-making-in-the-world-of-big-data/ (2012). Accessed 15 Apr 2013
14. Gainaru, A., Cappello, F., Kramer, W.: Taming of the shrew: modeling the normal and faulty behaviour of large-scale hpc systems. In: 2012 IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS), pp. 1168–1179. IEEE (2012)
15. Ghemawat, S., Gobioff, H., Leung, S.T.: The google file system. In: ACM SIGOPS Operating Systems Review, vol. 37, pp. 29–43. ACM (2003)
16. Hayler, A.: 'big data' applications bring new database choices, challenges. http://www.computerweekly.com/feature/Big-data-applications-bring-new-database-choices-challenges (2012). Accessed 15 Apr 2013
17. Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A.D., Katz, R., Shenker, S., Stoica, I.: Mesos: a platform for fine-grained resource sharing in the data center. In: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation, pp. 22–22. USENIX Association (2011)
18. Hindman, B., Konwinski, A., Zaharia, M., Stoica, I.: A common substrate for cluster computing. In: Workshop on Hot Topics in Cloud Computing (HotCloud), vol. 2009 (2009)
19. IBM Omnibond, X.: Big Data implementation: Hadoop and beyond. http://www.datanami.com/whitepapers/ (2013). Accessed 15 June 2013
20. Inc., G.: Bigquery, Official Website. https://developers.google.com/bigquery/ (2013). Accessed 15 June 2013
21. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks. ACM SIGOPS Oper. Syst. Rev. **41**(3), 59–72 (2007)
22. Krishnan, S.: Programming Windows Azure. O'Reilly (2010)
23. Lämmel, R.: Googles mapreduce programming modelrevisited. Sci. Comput. Program. **70**(1), 1–30 (2008)
24. Markoff, J.: Google cars drive themselves, in traffic. N.Y. Times **10**, A1 (2010)
25. Metz, C.: Meet the Data Brains Behind the Rise of Facebook. http://www.wired.com/wiredenterprise/2013/02/facebook-data-team/ (2013). Accessed 14 July 2013
26. Neumeyer, L., Robbins, B., Nair, A., Kesari, A.: S4: Distributed stream computing platform. In: 2010 IEEE International Conference on Data Mining Workshops (ICDMW), pp. 170–177. IEEE (2010)
27. Noseworthy, G.: Infographic: Managing the Big Flood of Big Data in Digital Marketing. http://analyzingmedia.com/2012/infographic-big-flood-of-big-data-in-digital-marketing/ (2012). Accessed 14 Apr 2013
28. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: a not-so-foreign language for data processing. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, pp. 1099–1110. ACM (2008)
29. Paskaleva, K.A.: Enabling the smart city: the progress of city e-governance in europe. Int. J. Innov. Reg. Dev. **1**(4), 405–422 (2009)
30. Patterson, D.A.: The data center is the computer. Commun. ACM **51**(1), 105–105 (2008)
31. Pavlo, A., Paulson, E., Rasin, A., Abadi, D.J., DeWitt, D.J., Madden, S., Stonebraker, M.: A comparison of approaches to large-scale data analysis. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, pp. 165–178. ACM (2009)
32. Pierre, G., Stratan, C.: Conpaas: a platform for hosting elastic cloud applications. IEEE Internet Comput. **16**(5), 88–92 (2012)

33. Pike, R., Dorward, S., Griesemer, R., Quinlan, S.: Interpreting the data: parallel analysis with sawzall. Sci. Program. **13**(4), 277–298 (2005)
34. Power, R., Li, J.: Piccolo: building fast, distributed programs with partitioned tables. In: OSDI, pp. 293–306 (2010)
35. Raicu, I., Foster, I.T., Zhao, Y.: Many-task computing for grids and supercomputers. In: Workshop on Many-Task Computing on Grids and Supercomputers, 2008 (MTAGS 2008). pp. 1–11. IEEE (2008)
36. Roush, W.: Facebook Doesnt have Big Data. It has Ginormous Data. http://www.xconomy.com/san-francisco/2013/02/14/how-facebook-uses-ginormous-data-to-grow-its-business/2/ (2013). Accessed 14 July 2013
37. Schatz, M.C.: Blastreduce: High Performance Short Read Mapping with Mapreduce. University of Maryland. http://cgis.cs.umd.edu/Grad/scholarlypapers/papers/MichaelSchatz.pdf
38. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Zhang, N., Antony, S., Liu, H., Murthy, R.: Hive-a petabyte scale data warehouse using Hadoop. In: 2010 IEEE 26th International Conference on Data Engineering (ICDE), pp. 996–1005. IEEE (2010)
39. Tudoran, R., Costan, A., Antoniu, G.: Mapiterativereduce: a framework for reduction-intensive data processing on azure clouds. In: Proceedings of Third International Workshop on MapReduce and Its Applications Date, pp. 9–16. ACM (2012)
40. Vrbić, R.: Data mining and cloud computing. JITA—J. Inf. Technol. Appl. (Banja Luka)-APEIRON **4**(2), 75–87 (2012)
41. Waas, F.M.: Beyond conventional data warehousingmassively parallel data processing with greenplum database. In: Business Intelligence for the Real-Time Enterprise, pp. 89–96. Springer, Berlin (2009)
42. Wampler, D.: Programming trends to watch: logic and probabilistic programming. http://thinkbiganalytics.com/programming-trends-to-watch-logic-and-probabilistic-programming/ (2013). Accessed 18 Apr 2013
43. Warneke, D., Kao, O.: Nephele: efficient parallel data processing in the cloud. In: Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers, p. 8. ACM (2009)
44. Yang, H.c., Dasdan, A., Hsiao, R.L., Parker, D.S.: Map-reduce-merge: simplified relational data processing on large clusters. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, pp. 1029–1040. ACM (2007)
45. Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, Ú., Gunda, P.K., Currey, J.: Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language. In: OSDI, vol. 8, pp. 1–14 (2008)
46. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, pp. 10–10 (2010)
47. Zaharia, M., Konwinski, A., Joseph, A.D., Katz, R., Stoica, I.: Improving mapreduce performance in heterogeneous environments. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, pp. 29–42 (2008)