Chapter 1
Introduction to Big Data

Amir H. Payberah and Fatemeh Rahimian

The amount of data generated during the last few years has been unprecedented. This is not only due to the prevalence of online social networks and the ubiquitous devices connected to the Internet, but also as the result of the advances in technology across other fields, for instance, whole genome sequencing. Hence, it is fair to say that we are living in the era of *big data*. Big data refers to large datasets or data flows that have outpaced our capability to store and process, and cannot be analyzed by traditional means. More specifically, challenges arise mainly due to one or several of the following reasons:

- *Volume*: when we encounter massive data in size, e.g., data from crawling the web, or genome sequencing data, traditional storage and processing systems fall short. We, thus, need to build new systems, techniques, and algorithms that efficiently store, retrieve, and process huge volumes of data.
- *Velocity*: big data is not only about the size. High rate of data generation is also important. For example, data generated in Twitter or communication networks come in form of continuous streams of data at a very high rate. Many systems require to analyze this kind of data in real-time.
- *Variety*: sometimes data comes from multiple sources and in a variety of forms, for example as a combination of structured, semi-structured, and unstructured data. It is, therefore, important to have systems that handle diverse data models without compromising performance.

In the presence of these challenges, traditional platforms fail to show the expected performance, and thus, new systems for storing and processing large-scale data are crucial to emerge. In this chapter we explore some of the new trends of technology for handling big data.

1.1 Big Data Platforms: Challenges and Requirements

A big data platform should provide means to efficiently store, retrieve, and process massive amount of data. One of the main challenges a big data platform should address is *scalability*. More specifically, the platform should allocate as much resources as required for handling big data. There are two possible solutions to make a system scalable: (i) to *scale up* (or *scale vertically*), by adding more resources to a single machine, or (ii) to *scale out* (or *scale horizontally*), by adding more machines in a network and use all their collective resources. Buying an extremely strong machine for scaling up is probably less challenging, but it is very costly. More importantly, you can scale up a system only to a certain degree, i.e., there is a limit in how much resources you can add to a single machine, and this limit is far less than what most big data processing applications require. In contrast, exploiting the collective resources of a network of commodity machines is an economically and technically attractive solution, and thus, scaling out is the approach taken by almost all the existing platforms. Nevertheless, due to the distribution of data and computation over a network, new challenges and requirements arise:

- *Fault tolerance*: one or several machines may fail while running a job. Assume a machine can stay up for 1000 days. If there are 1000 machines in a network, we expect to observe one failed machine per day, on average. When there are millions of machines in a network, like in Google sites, we may have 1000 machine failures per day. It is, therefore, crucial for the the platform to be resilient to the failures.

- *Transparency*: while resources of a platform are distributed, it is widely agreed that users should get an illusion of working with one single machine. More precisely, the details of resource management, including resource allocation and load balancing, should be hidden from an ordinary user of the platform. This is one of the requirements of any big data processing platform.

- *Parallel programming model*: traditional programming models assume that code, data and all the required resources for executing the code (e.g., CPU and memory) are available locally. This assumption is not valid anymore in horizontally scalable platforms. In the new model, data and/or operations should be parallelized, so that different parts of the data can be processed in parallel. Moreover, since transferring large amounts of data over network is costly, it is often the code that is sent over to where the data is stored. This paradigm shift calls for the development of many new parallel and distributed algorithms.

- *Shared-nothing communication model*: processes can communicate over a network in three different ways: via storage, memory or network. These models are known as shared-storage, shared-memory, and *shared-nothing*, respectively [1]. For scalability reasons, the shared-nothing architecture has become the de-facto communication model in building big data platforms.

Currently there exist several big data platforms that provide the above features. The diversity of these platforms can make it difficult to choose the best one for carrying out a task. Some platforms are designed for a specific type of processing, for example GraphLab [2] for graph processing and Storm [3] for stream processing, while some others are more generic and handle a wider range of processing types. Example of such platforms include MapReduce [4], Spark [5] and Flink [6]. While

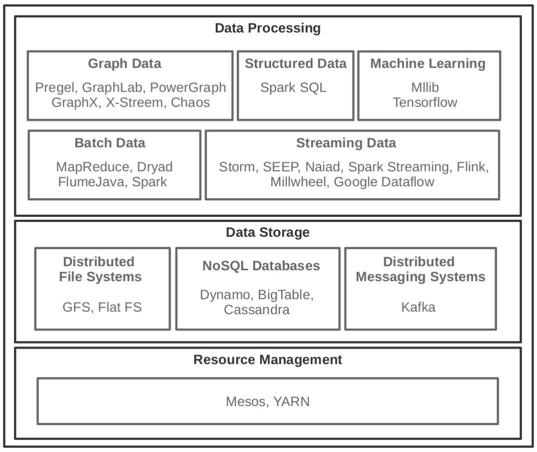

Figure 1.1: Big data platforms stack.

the overall architecture of these platforms share many common features, the platforms themselves can be integrated in a stack, depicted in Figure 1.1, which consists of the following layers:

- *Resource management*: this layer contains platforms that are used to manage resources of a cluster and share them among the platforms in the upper layers.
- *Data store*: the platforms in this layer are used to store and retrieve massive data. They includes distributed file systems that maintain data on distributed disks, messaging system for handling real time data, and databases to maintain structured data at scale.
- *Data processing*: this layer contains the platforms for parallel processing of data across a large number of commodity computers. These platforms are categorized into a few subgroups, based on their target application and input model, for example for batch data, streaming data, graph data, structured data, or for higher level analysis, e.g., machine learning algorithms.

Due to lack of space, we chose to skip the platforms in the resource management layer. We will, however, explore some of the well-known platforms in the two top layers of Figure 1.1 that answer two main questions: (i) how to *store* big data, and (ii) how to *process* it.

1.2 How to Store Big Data?

When the size of data exceeds the capacity of one disk, we have to use multiple disks in a distributed environment. To build a distributed storage system, we need to take into account the nature of data that we are going to store. We could be dealing with batch or streaming data, and the data could be structured or unstructured. Based on these characteristics and also on the target application, data can be stored in either a file system, a messaging system, or a database. In this section we will explain some of the well-known storage systems.

1.2.1 Distributed File Systems

In operating systems (OS), a *file system* refers to a collection of methods and data structures to store files on a disk and retrieve them. In Unix-like file systems, for instance, a file is divided into small data blocks, which are stored on a disk. The OS, then, uses a data structure, called *inode* to maintain the file's metadata, e.g., ownership and access mode, as well as the location of the file's data blocks on disk. The inode structure is originally designed for a single disk, and does not work over multiple and distributed disks. We, thus, need to design a *distributed file system* that makes it possible to store and retrieve files on/from distributed disks, without involving users in details and complexity of the system. Several distributed file systems have been designed and developed, e.g., GFS [7], FlatFS [8], and Ceph [9], among which GFS, and its open source implementation HDFS [10], are the most popular ones.

GFS and HDFS

In GFS, a file is split into a number of *chunks*. A chunk is a single unit of storage, which is transparent to users. Size of chunks is chosen relatively big (64MB or 128MB), compared to block size in OSs, to reduce the read/write time. From the architectural perspective, GFS has three main components: *master*, *chunk server*, and *client*. The master (similar to inode) stores metadata about files and the location of their chunks, while chunk servers store chunks as regular files on their local file systems. The clients, then, find the location of chunks by contacting the master, and continue the rest of operation, e.g., read and write, by communicating directly with the respective chunk server(s).

The GFS master maintains the file system namespace as a key-value table, with file full pathname as key and the metadata as value. It also manages the access control to files by acquiring a set of read/write locks on files in the namespace. For example, in the path `/foo/bar/test.txt`, the master can apply a read lock on internal nodes, e.g., `/foo` or `/bar`, to prevent the deletion or renaming of them and their descendant sub-trees. Similarly, it can apply a read/write lock on the leaf nodes, e.g., `test.txt`, to protect them from further read and write operations, while they are opened by one client.

In GFS, each chunk is replicated on a number of chunk servers to increase data reliability and availability. The master decides on replica placement, by placing the replicas on chunk servers with below-average disk usage. It also creates new replicas when the number of available replicas falls below a predefined threshold. To provide consistency among replicas of a chunk, one replica is designated as the *primary* for that chunk, and the other replicas are maintained as *secondaries*. The primary replica decides the update order, and the secondaries follow this order.

GFS does not provide POSIX-based APIs for interaction, but it provides functionalities to read, write and delete files. To read a chunk, the user application originates a read request and delivers it to a GFS client, who sends the request to the master. Upon receipt of a read request, the master responds with the address of replicas (over chunk servers). The client selects one of these chuck servers and sends it the read request. Finally, the chunk server sends the requested data back to the client, and the client forwards it to the application. Similarly, to write a chunk, the application sends a write request to a client, which in turn forwards the request to the master. Once more, the master replies with the address of existing replicas on the chunk servers. When the client receives this information, it pushes data to all the corresponding chunk servers, both primary and secondaries. The chunk servers keep the received data in their internal buffers, without writing them to their disk. When the client issues a write command, the primary serializes data instances, that is, it writes the updates to chunks in a specific order. It then sends the data instance order to the secondaries, so that they apply the update in that same order. The delete function, is however, a metadata operation, meaning that when a user calls it, the master just marks the name of the file as deleted, but the actual data will remain on disks. After a certain time, the master deletes the data of all the marked files.

Since all the metadata information about the file system is on the GFS master, the system can not work if the master fails. To make the system robust, the master state is also replicated on multiple machines. If the master fails, a new master takes over and continues from the latests replicated state.

1.2.2 Messaging Systems

Sometimes the complete data is not available in the beginning of a process, and instead, it is received as streaming data gradually over time. For example, a web server, as a data provider, continuously sends events every time someone requests a page. The consumers, then, can use this data for different purposes, e.g., to store in HDFS, trigger an alert, or send a notification email. A *messaging system* is a middleware that facilitates a near real-time asynchronous computation by decoupling all consumers works from the actual data provider services. When a new event takes place at a provider, messages are added to the messaging system, and consumers can read them based on their demands. Several messaging systems exist, e.g., Kafka [11], ActiveMQ [12], RabbitMQ [13], and Flume [14]. Among this list we will briefly explain Kafka.

Kafka

Kafka [11] is a distributed topic oriented log service, which was designed originally in LinkedIn. It categorizes feeds of messages into multiple groups, called *topics*, each containing a stream of messages of a particular type. Each Topic is divided into a number of *partitions*, each being an append-only and immutable file on disk. Messages generated by a producer to a particular topic partition are appended in the same order they are sent, and consumers see them in the same order they are stored.

To increase the reliability of the system, partitions of a topic are replicated on several servers, called *brokers*, where one broker becomes the leader of a partition, and all writes and reads are managed through it. Kafka uses Zookeeper [15] to manage its system on a cluster of machines. Zookeeper detects the addition and the removal of brokers and consumers, maintains the consumption relationship, and keeps track of the consumed offset of each partition.

1.2.3 NoSQL Databases

File systems store any type of data, whether it is structured or unstructured, but they provide no means to take advantage of the structure in data in the former case. This is where database systems can play an important role. Databases are built on top of file systems to deal with well formatted data and perform efficient read, write, update, and delete operations. Among the existing databases, relational database management systems (RDBMSs) are the dominant ones for maintaining structured data. RDBMSs guarantee certain properties, commonly known as the *ACID properties*, with the following descriptions: (i) *Atomicity*: either all or none of the operations in a *transaction* are executed, where a transaction is a single unit of work and consists of a sequence of operations in a database, (ii) *Consistency*: database should be in a consistent state before and after a transaction, (iii) *Isolation*: uncommitted changes in the databases should not be visible to other transactions, and (iv) *Durability*: changes should be written to a disk before a transaction is marked as committed, so that any updated data could be later recovered in case the system fails.

With the emergence of big data in various domains, for example over the Web 2.0 applications, data management technologies are entering a new phase. The big data applications have special demands, such as *scalability* and *availability*, which are not necessarily in-line with the ACID properties provided by RDBMSs. For example, when we are dealing with a high rate of read/write operations, treating each operation as a transaction and locking the data to provide ACID guarantees, may hinder the scalability of the system. One way out of this problem is to relax some of the unnecessarily strong properties, for instance consistency and isolation. In fact, a new set of properties have been defined for such scenarios. These properties, known as *BASE properties*, are introduced to trade consistency and isolation of ACID properties for achieving scalability and availability. The BASE properties are: (i) *Basic Availability*: faults may happen, but they should not obstruct the functioning of the whole system, (ii) *Soft-state*: different copies of a data piece may be inconsistent, and (iii) *Eventually consistent*: all copies of a data piece eventually become consistent at some point in future, if no more updates happen to that data piece. The BASE properties have become the baseline in all the emerging databases, knowns as *NoSQL (Not Only SQL)* databases that deal with big data.

To put the two sets of properties in perspective, it is perhaps useful to recall the famous *CAP theorem*, which states that in any distributed system, it is impossible to provide consistency (C), availability (A), and partition tolerance (P) properties all at the same time. In other words, a distributed database system can have only two of these properties simultaneously. While most RDBMSs have chosen to provide consistency and availability, without providing partition tolerance, NoSQL databases are always partition tolerant, but provide either consistency or availability, not both at the same time.

NoSQL databases can have different data models, that is they can store data in different ways. There exist four popular data models in use, namely, *key-value*, *column-based*, *document-based*, and *graph-based*. Key-value data model is the simplest data model, where data is stored in form of pairs of key and value, and values could be any arbitrary data. Column-based data model enhances the key-value model by adding some schema to values. The document-based databases are similar to column-based store, except that values can have a flexible schema (e.g., XML, or JSON), instead of a fixed schema. Finally graph databases model data and its interdependencies as a graph, and store it in form of graph nodes, edges, and properties.

In this part we introduce two different NoSQL databases, Dynamo [16] and BigTable [17], where the former is a key-value store that provides P and A properties, and the latter is a column-based storage that provides P and C properties.

Dynamo

Dynamo [16] borrows the idea of *consistent hashing* [18] from Distributed Hash Tables (DHTs) for partitioning and distributing data across multiple machines. Each machine is given an *identifier (id)*, using a hash function, and the machines are ordered along a ring by their ascending ids. The same hash function is also applied on data to give each data item an id in the same id space. Each machine, then, stores data items with ids between its own node id and its *predecessor* id in the ring. The predecessor of a node B would be a node A, whose id is the previous id anti-clockwise in the ring before the B's id. In this case, node B is the *successor* of node A. To achieve high availability and durability, Dynamo replicates data on multiple machines, listed on a *preference list* per data item, which are usually the n successor machines of an id along the ring.

Although consistent hashing is an efficient way to distribute data over machines, it may end up with imbalanced load on machines, due to several reasons, such as non-uniform distribution of data ids (keys) over the ring, various popularity and hit rate of the keys, or the heterogeneous power of machines. To overcome these challenges, Dynamo uses *virtual nodes*, meaning that each physical machines picks multiple random ids, where each id represents a virtual node. Hence, we can assume that each physical machine runs multiple virtual nodes over different parts of the ring, where each virtual node covers the range of keys between its id and its predecessor virtual node id.

To read and write data, Dynamo provides two APIs: `get`, which returns a single item or list of items with conflicting versions, and `put`, which stores an item under a given key. These operations are handled by a *coordinator* for each item, which is the first node on its preference list. As mentioned earlier, Dynamo does not guarantee strong consistency, but provides eventual consistency, which enables asynchronous updates of data items. More precisely, multiple versions of a data item may exist in the system, but replicas of each item eventually become consistent. Dynamo tracks the causality of events over different replicas, and if it identifies an order among them, it replaces the older version of replica with the new one, otherwise it raises a conflict, in which case reconciliation is required. A conflict may happen due to node or network failures. If multiple versions of a data item exist, the system delegates the reconciliation to users. Such a scenario can happen in online shopping, for example, when a user finds inconsistent shopping baskets in her profile. The system never refuses to add new items to the basket, but a user may find some already removed items, back in the basket again.

Machines can be added to or removed from Dynamo by an administrator. After new machines are added/removed, the membership change is propagated in the system using a gossiping protocol [19], such that eventually all machines acquire a consistent view of the system. When a new machine joins, it gets an id and thus, a key range for which it is responsible. Then, the data items that fall into that range are transferred to the new machine. For example, assume a new machine X is added to the system between two existing machines A and B, where B was the successor of A and responsible for the key range $[A,B)$. After adding X, the data items for the key range $[A,X)$ should be transferred to X, and B would be responsible for the new key range $[X,B)$. When a machine is removed, for example the newly added X, a reverse process will take place, during which data items on X are transferred to its successor B. This is how the number of machines in Dynamo can dynamically change.

BigTable and HBase

BigTable [17], introduced by Google, is another NoSQL database. While BigTable is built on top of GFS, the open source version of it, that is HBase [20], is part of the Hadoop ecosystem and is built over HDFS. BigTable uses a column-based data model to store data. A *table* is the highest abstraction to store data. Each table consists of multiple rows, where each row has one or more columns. Rows are ordered lexicographically by their key. A group of columns with the same type can build a *group family*, which are the basic units of access control. Each cell in the table can have multiple values, distinguished by their timestamps. When a table becomes too large, the system splits it into *tablets*, which are contiguous rows stored together.

BigTable has three main components: *master server*, *tablet server*, and *client library*. In each cluster there exist only one master server, which assigns tablets to tablet servers. The master server also balances the load among tablet servers, and conducts garbage collection of useless files in GFS. Moreover, it handles the changes to the schema, e.g., creates new tables or adds new column families. Management of tablets are done by the tablet servers. Multiple tablet servers exist in a cluster, and they can be added or removed dynamically. Each tablet server is in charge of a set of tablets and all the read and write operations that apply to those tablets. Each tablet is assigned to only one tablet server. Note that the data of tablets are stored in GFS, and tablet servers only handle the read and write requests for their assigned tablets. Since files are replicated in the GFS layer, there is no need to replicate tablets separately. Client libraries provide methods to communicate with the master and tablet servers and cache the tablet locations. Clients can work with BigTable through these libraries.

BigTable takes advantage of other existing platforms internally. For example, it uses GFS to store log and data files, and Chubby lock service [21] to manage the deployed system. Chubby is responsible for the following tasks: (i) to ensure only one master is active in cluster, (ii) to store the location of the *root tablet* that contains the location of all other tablets, (iii) to discover tablet servers, (iv) to store tables' schema, and (v) to store access control lists. When a master starts, it communicates with Chubby and grabs a lock to prevent any other master claiming the system. It, then, gets the list of available tablet servers from Chubby and communicates with them to discover their already assigned tablets.

BigTable uses a 3-level hierarchical structure to maintain the address of tablets: (i) Chubby stores a file that contains the address of a metadata tablet, called *root tablet*, (ii) the root tablet contains the location of all other metadata tablets, and (iii) each metadata tablet maintains the address of a set of user tablets. Each tablet, internally, is divided into a number of *SSTables*, which are the fundamental components of BigTable for storing data. An SSTable is a set of immutable sorted key-value pairs, stored as a file in GFS.

When a user commits some update to a tablet, first the commit logs are stored in GFS. Then, the responsible tablet server for that tablet keeps the most recent updates in an in-memory structure, called *memtable*. When the size of a memtable exceeds some threshold, it is written to an SSTable, and consequently to GFS. This can eventually result in having a large number of SSTables in GFS, and thus, the system periodically merges the SSTables of each tablet into a single SSTable, to optimize the disk usage. To read data from a tablet server, both memtable, which contains the latest updates, and the sequence of recent SSTables are used.

BigTable guarantees strong consistency, because each tablet is managed by one tablet server only, and all concurrent queries for a tablet are serialized in that tablet server. However, if a tablet server fails, the availability of its part of data is violated until a new server is assigned. In other words, BigTable provides consistency, but can not guarantee availability.

1.3 How to Process Big Data?

The next big challenge while dealing with massive data, is how to process it. Various platforms and tools have been recently developed for this purpose, and choosing the right tool is essential. The existing tools can be categorized based on the kind of data they process, for example batch data, streaming data, graph (linked) data, and structured data. In this section we explore some of the state-of-the-art tools from each of these categories.

1.3.1 Batch Data Processing Platforms

Processing batch data, also known as *data-at-rest*, is the traditional way of data processing. Building a single machine system for batch processing is simple and well studied since when the first generation of computers emerged. When dealing with batch data, we know that all the data is available at the processing time, but in case of big data, it may be too big to be loaded into the memory all at once. Hence, when the size of data exceeds the capability of one machine, then new solutions are required.

To provide a practical example, assume there is a text file and the goal is to count the number of distinct words in this file. Also, assume the size of the file is small enough to be loaded into the memory of one machine. In this case, a simple bash script command can count the number of words:

```
words(file) | sort | uniq -c
```

where `words(file)` splits the words of the given file by space and returns a list of words. However, if the file does not fit in the memory of one machine, the above script does not work any longer. A possible solution to scale up the system is to divide the file and distribute it across several machines and process them in parallel. However, new challenges arise with such a system, including parallelization, fault tolerance, data distribution, and load balancing.

MapReduce

MapReduce [4] is one of the first batch data processing systems that addressed the above challenges, while providing users with a new programming model that enables them to implement their code easily. In other words, MapReduce is both (i) a programming model for big data processing, inspired by functional programming, and (ii) an execution framework to run parallel algorithms on clusters of commodity machines.

Programming in MapReduce model boils down to writing two main functions: a *map* function that processes data and generates a set of intermediate key-value pairs, and (ii) a *reduce* function that aggregates all the intermediate values associated with the same intermediate key. There is also a *shuffle* step that takes place between the execution of these two functions. During the shuffle step, the key-value pairs that are generated by the map function, are sorted and prepared for the reduce function.

To implement the "word count" – the process of listing the words accompanies with the number of their occurrences in a file – example using this model, the following three steps can be performed: (i) `words(file)` extracts words from `file`, (ii) `sort` shuffles and sorts the words, and (iii) `uniq -c` aggregates the intermediate results and generates the final output. This code can be perfectly modeled with MapReduce, where each command corresponds to one of the phases of MapReduce. If the sample input file contains `Hello World, Hello Life`, then the map function reads the words and for each one generates a key-value pair with value 1, e.g., `(Hello, 1)`, `(World, 1)`, `(Hello, 1)` and `(Life, 1)`. The shuffle phase between map and reduce phase creates a list of values associated with each key, e.g., `(Hello, (1, 1))`, `(World, (1))` and `(Life, (1))`. Finally, the reduce function sums up the counts per key and generates the final result, e.g., `(Hello, 2)`, `(World, 1)` and `(Life, 1)`. Note that the user needs only to implement the map and reduce functions, and the system takes care of the shuffle phase.

An important notion here is that, while the code is very small, the data can be big and possibly distributed over multiple machines in a network. A traditional computation model will move the data over the network to be read and processed by the code. In contrast, the MapReduce computation model suggests that we keep data where it is, and instead move the computation close to data. The small piece of code can then be executed in parallel on each machine, and the result will be aggregated and reported. More specifically the following steps are taken to execute a program in MapReduce:

1. The input files are read and divided into a number of *splits*. The size of splits is typically the same as the size of chunks in HDFS.

2. The MapReduce library in the user program, then, sends a copy of the program to each of the machines, among which one becomes the *master*, and the others become *workers*. The master assigns tasks (map or reduce) to the workers, who become mappers and reducers, accordingly.

3. Each mapper takes a set of splits as input and performs the map function on them. The result of the map function is generated as intermediate key-value pairs, which are buffered in the memory of the mapper.

4. Each mapper periodically writes the buffered data to its local disk, and sends their addresses to the master. Then, the master forwards these addresses to the reducers.

5. Each reducer reads the corresponding intermediate data from the local disks of the mappers. When a reducer reads all the required key-value pairs, it sorts and groups them by their keys.

6. Each reducer, then, iterates over the list of intermediate keys and their corresponding values, and performs the reduce function on them. The result of reduce functions are appended to the final output file in HDFS.

7. When all map and reduce tasks have been completed, the master informs the user program that the final result is ready.

The master monitors workers liveness via periodic heartbeats. If it detects the failure of an in-progress map or reduce task, it re-execute it (possibly on a different worker). If it detect a completed map task has failed, it again need to re-execute the map task, because the output is stored on the local disk of the failed mapper. However, if a reducer with a completed task fails, the master does not re-execute the task, because the output is stored in HDFS. The state of the master is periodically check-pointed. Hence, upon failure, a new master starts and resumes the work from the last check-pointed state.

Spark

Although MapReduce facilitates an easy implementation of batch data processing over a cluster, it is very rigid in nature and can not be used for building complex, interactive or iterative programs. Sometimes adding only a little complexity can render the whole MapReduce model infeasible. For example, let us add a few steps to the word count example:

```
words(doc.txt) | grep | sed | sort | awk
```

This is a job that requires more than one map and reduce round, and each two consecutive rounds can only communicate through HDFS. That is, the reducer of one round writes the result in HDFS, and the mapper of the next round reads that data from HDFS. However, reading from and writing to HDFS is a slow process. To overcome this problem we need to reduce the interaction with HDFS as much as possible, for example by keeping the intermediate results in memory, when there are multiple consecutive rounds of map and reduce functions. Replacing a stable storage with volatile memory is challenging, and the question is how to make such a memory model efficient and fault tolerant.

Spark [5] provides an answer to this question. It is a batch processing engine for massive data, which exploits in-memory processing by presenting a distributed memory abstraction, called *Resilient Distributed Datasets (RDD)*. RDD is an immutable collections of objects spread across a cluster. An RDD is divided into a number of *partitions* (atomic pieces of information), where each partition can be stored on a different machine in a cluster.

Spark works based on the master-worker model. The main program, the *driver*, runs on a master machine and coordinates the execution of the whole application. When a Spark application is executed, the driver connects to the cluster manager and acquires *executors* on worker machines to run tasks and store data (one or more partitions of RDDs). The driver, then, sends the application code, as well as tasks to the executors. The entry point to Spark functionalities is through a `SparkContext` object in the driver that defines how Spark can access the cluster, e.g., run locally, run as a standalone cluster, or run on cluster via a resource management system, such as Mesos [22] or YARN [23].

There are two types of operators that can be applied on RDDs: *transformations* and *actions*. Transformations are *lazy* operators that are applied on RDDs and create new RDDs. They are called lazy, because they do not compute the result right away. Instead, they build a chain (graph) of operations over RDD, called *lineage graph*. Actions, on the other hand, lunch a computation on RDDs and return a value. When an action is called on a RDD, all the transformations in its lineage graph are executed and then the final result is computed. More specifically, upon calling an action, RDDs are broken down into multiple partitions and are loaded by the Spark executors on worker nodes. Then, transformations are executed and finally the result are calculated. When multiple actions are called on an RDD, all the transformations in its lineage graph are recomputed per action. To reduce the overhead of recomputation however, the transformed RDDs can be *cached* in memory. The caching, if needed, should be explicitly done by the programmer.

The lineage graph is also used to recover from failures in an efficient way. Unlike MapReduce that replicates data to make the system resilient, Spark keeps track of the lineage information, by which it can reconstruct the lost partitions. If a partition fails, Spark backtracks on the lineage graph until it finds a correct partition, and then recomputes the lost partitions of RDDs. If a RDD becomes unavailable, all its missing partitions are recomputed in parallel. If a task fails, it is re-executed on another machine, providing that its parent RDDs on the lineage graph are available.

1.3.2 Streaming Data Processing Platforms

Some applications need to process streams of live data and provide results in real-time. Wireless sensor network services, traffic management systems, and stock markets are examples of such applications. Stream Processing Systems (SPS) are a group of platforms that process such streaming data [24]. In contrast to batch processing systems and Database Management Systems (DBMS), which are used to analyze data-at-rest, a SPS processes *data-in-motion*. Typically, batch processing systems and DBMSs store and index data before computation, and process them only when explicitly asked by users. However, a SPS processes data as it arrives, without having to store it persistently.

A SPS receives streaming data as an unbounded sequence of individual data items, called *tuples*. A tuple is the atomic data item in a streaming data, which is equivalent to a row in table. The tuples can be either structured in a predefined schema, semi-structured with self-describing tags (e.g., XML), or totally unstructured in custom formats (e.g., video and/or audio).

The programming model for a SPS is normally based on defining *jobs* in form of *dataflows* to represent the *logical plan* of the work. A dataflow is a *directed acyclic graphs (DAG)* composed of data sources, *processing elements (PE)*, and data sinks. A PE is the basic functional unit in a dataflow that reads some input tuples, applies a specific function on them, and outputs new tuples.

Two fundamental questions regarding the dataflow programming model are (i) how to compose a dataflow, and (ii) what functions to use. Dataflow composition is the process of creating a DAG associated with a job. DAG composition can be *static* or *dynamic*. If all the PEs and their relation in the DAG are known in advance, they can be connected statically, otherwise the dynamic composition is used. The PEs that are put in a DAG in this step are higher order functions that belong to one of the following operation categories: (i) *aggregation*, to collect and summarize a subset of tuples, (ii) *merge/split* to combine/partition input streams, (iii) *logical* and *mathematical* operations, (iv) *sequence manipulations*, to reorder or delay tuples, or (v) any other custom data manipulations, e.g., data mining algorithms. Each of these categories include many different functions, and thus, the next step is to decide which function should be used inside each PE.

A PE can be either *stateless* or *stateful*. In a stateless model, a PE processes tuples independent of each other and then forgets about them; whereas in a stateful model, a PE is a *synopsis* of the already received tuples, meaning that it maintains an internal state with the footprint of the processed tuples. In this case, a PE also keeps a subset of the most recent tuples in a buffer, namely a *window*. There exist two popular window models: *tumbling* and *sliding*. Both models keep a certain number of tuples, defined by the window size. When the buffer is full, a tumbling window will remove all the buffered tuples at once, while a sliding window only removes the oldest ones from the buffer. The tumbling window model is usually used for batch operations, while the sliding window model fits better in scenarios with incremental operations.

More specifically, the semantics of a window model is defined by its *eviction policy* and *trigger policy*, where the eviction policy determines the properties of tuples that are to be removed, and the trigger policy defines when the buffered tuples should be processed. In general four different policies are available: (i) *count-based*, which defines the maximum number of tuples the buffer can hold (for an eviction policy) or the number of tuples that should be received before the tuples can be processed (for a trigger policy), (ii) *delta-based*, which is specified by a delta threshold in a tuple attribute, for eviction or trigger, (iii) *time-based*, which defines a time interval for eviction or trigger, and (iv) *punctuation-based*, which triggers processing or eviction of tuples, upon receipt of a punctuation. Any combination of these policies can be used independently for eviction and trigger. For example, a count-based eviction policy could co-exist with a time-based trigger policy.

The dataflow that a user defines is a *logical plan* that should be converted to a *physical plan* at run time and deployed over a cluster. Vertices and edges of a logical plan correspond to PEs and their connections, respectively. Whereas, in a physical plan vertices represent the operating system processes, and edges denote the data communication medium (e.g., network connection and/or shared memory). The physical plan is not unique and the transformation task is not straightforward. A decent physical plan takes into account the workload of each PE and the amount of data transfer between different PEs, when partitioning the logical plan and deciding if a partition or a set of PEs should be located on a single machine or multiple ones. These are however, similar to the challenges of parallelization in general.

Parallelization enables the SPSs to remain efficient with the increasing number of queries and the high rate of incoming data. There are different ways to parallelize a SPS. The first approach is *pipelined parallelization*, where sequential PEs of a dataflow run concurrently on different tuples of a stream. For example, if A and B are sequential PEs, represented as ($A \rightarrow B$), then B can start processing a `tuple1`, as soon as A completes processing it and moves on to process `tuple2`. The second model is *task parallelization* in which, independent PEs are executed concurrently on the same or distinct tuples. For example, if A and B are independent PEs, they can run in parallel on the same tuple, e.g., `tuple1`. *Data parallelization* is the third model, where the same PE runs in parallel on different parts of a tuple. For example if `tuple1` is a big data item, it can be divided into a number of parts, and different instances of a single PE, e.g., A, can be executed concurrently on different parts of the tuple. In the data parallelization model, the incoming tuples can be distributed randomly between PEs, or they can grouped by some keys and divided between PEs, or all tuples can be sent to all PEs.

Since failures are inevitable in a distributed system, data recovery becomes an important challenge for any SPS. A popular technique for avoiding data loss is *rollback recovery*, which can benefit from either an *active backup*, *passive backup*, or *upstream backup*. In the active backup, a backup node is associated with each processing node (called *primary*), and the same input is given to both primary and backup nodes. However, the output of the backup node is logged and is not sent downstream.

Once the primary fails, the backup node takes over and sends the logged tuples to all downstream nodes and remains active afterwards. In the passive backup, the state of each node is periodically checkpointed in a shared storage. If a node fails, it will be replaced by a new node to take over from the latest checkpoint. Finally, in the upstream backup, upstream nodes (the parent node in DAG) store and keep the tuples until the downstream nodes acknowledge that the tuples are not needed any longer. If a node fails, a new node takes over by rebuilding the latest state of the failed node from the logged tuples at the upstream node.

In the rest of this section, we will explain three stream processing systems, Spark Streaming [25], Storm [3] and Flink [6]; Spark Streaming uses a *mini-batching* processing model, while the other two use a *tuple-at-a-time* processing model. In the mini-batching processing model, the streaming data is divided into small batches, and the streaming process is run as a series of deterministic computations over the batches. In the tuple-at-a-time processing model, stateful PEs process every incoming tuple, update their internal state, and emit new tuples.

Spark Streaming

Spark Streaming [25] is a SPS built on top of Spark that runs a streaming computation as a sequence of small and deterministic batch jobs. The incoming streaming data is divided into batches of n seconds, and each batch of data is treated as one RDD. A continuous sequence of RDDs is called *Discretized Stream* or *DStream*. DStream supports different operations, including standard RDD operations (such as `map` and `join`), as well as other operation specifically developed for DStream (such as window operations). When an operation is applied on a DStream, it will be applied on all its RDDs, and the final result would be a new DStream.

Spark Streaming supports the sliding window model and allows to apply a transformation over a set of RDDs collected in a window. A sliding window is defined by two parameters: *window length* that declares the size of window in time, and *slide interval* that defines how much a window should slide every time. Note that if we need to apply a function over all the received RDDs, then the sliding window is not enough. In this case, we should checkpoint and maintain the computation state, while continuously updating it with new incoming data. To enable checkpointing, user should create a directory in a reliable storage where the checkpointed states will be saved. Given the checkpointed data, user can apply a function over the state as well as on the new incoming data.

Spark Streaming architecture follows a master-worker model, where the master keeps track of DStream dataflow graph and schedules tasks on worker nodes, and workers keep partitions of RDDs and execute tasks. Moreover, workers receive data from client libraries or load them periodically from an external storage. The master, then, tracks the location of data items and helps clients to find the required data. To make the system fault tolerant, Spark Streaming takes advantage of the *lineage graph* used in the core of Spark by remembering the sequence of transformations over RDDs. If some data is lost due to a worker failure, it can be recomputed using the parent RDDs in the lineage graph. Moreover, the input data stream is replicated in memory of multiple worker nodes, so that in the worst case, when all the transformations should be recomputed from scratch, the original data is accessible.

Storm

While Spark Streaming is a non-native SPS, meaning that it discretizes the input stream into mini-batches, and applies short-lived batch tasks over them, Storm [3] and Flink [6] are two native SPSs. In these systems we have long-lived task execution, where each task maintains its own state. Storm is a distributed SPS for real-time processing of streaming data. There are two types of PEs in Storm: *spouts* as sources of streams, and *bolts* that contain the main computation functions. Each bolt receives tuples from spouts and/or other bolts, processes them, and emits new tuples. In the Storm terminology, the DAG of spouts and bolts is called *topology*. To execute a topology, Storm runs spouts and bolts in parallel on different machines of a cluster. It is through the data and task parallelization models that Strom provide scalability.

Storm provides two types of *delivery* semantic guarantees: *at-most-once*, where each tuple is either processed once, or dropped if a failure happens, and *at-least-once* (also called *reliable* processing), in which each tuple is processed at least once even if failures happen. To guarantee the reliable delivery, Storm uses a number system level bolts, called *acker bolts*, which keep track of the tuples of every spout in a topology. When a bolt successfully executes its function on a received tuple, it notifies the acker bolt by sending an ack message to it. When the acker bolt receives an ack message for all tuples in a *tuple tree*, it sends a final ack to the spout that emitted the tuple. A tuple tree, refers to all the tuples emitted by subsequent bolts starting from a spout tuple. A spout also assigns a timeout for each tuple, and the acker bolt keeps track of these timeouts. If the ack message for a tuple does not arrive by the timeout, the tuple is considered to has failed, and thus, it is replayed by the spout.

The Storm cluster consists of two main components: (i) one master, called *nimbus*, that distributes and coordinates the execution of topologies, and (ii) a number of worker nodes that carry out the actual stream processing. A worker node executes one or more *worker processes*. Every worker process, in turn, runs one or more *executors*, each containing one or more tasks (spouts or bolts). Each worker node also runs a *supervisor* that receives assignments from nimbus and spawns worker processes for those assignments. The supervisor periodically contacts nimbus and inform it about the topologies the worker node is currently running, as well as the available resources for running more assignments and topologies. To coordinate the interaction between nimbus and the supervisors, Storm takes advantage of Zookeeper [15] coordination service. Zookeeper also provides fault tolerance, by maintaining the state of both nimbus and supervisors.

Flink

Flink is a distributed dataflow processing system that unifies stream and batch processing. Similar to previous systems, a job in Flink is defined as a DAG of PEs and their connections. In addition to the basic transformations, e.g., `map`, `reduce`, and `filter`, Flink provides binary stream transformations, e.g., `coMap` and `coReduce`, flexible window operations, and native iterations. It also supports several different windowing policies, including time-based, count-based, and delta-based windows.

Flink uses a master to schedule tasks, coordinate checkpoints, and perform recovery in case of failures. Jobs are submitted to the master in form of a dataflow graph (*job graph*). The master first transforms the job graph to an *execution graph*, which consists of information on job scheduling along with the tasks. Then, it sends the tasks to the workers, which perform the real computations by running one or more processes that carry out the assigned tasks.

As we explained, the fault tolerance in Spark Streaming is *coarse-grained*, based on RDD recomputation. On the other hand, the recovery in Storm is *fine-grained*, as it keeps track of each tuple individually. The fault tolerance in Flink is something in between: instead of asking an acknowledgement per tuple, a sequence of tuples are acknowledges together. Flink uses asynchronous *barrier* snapshotting for globally consistent checkpoints, inspired by Chandy-Lamport snapshot algorithm [26]. In this model, data sources periodically inject checkpoint barriers into the data stream that flows through the connections of the DAG. Upon receipt of a barrier at a PE, it emits all the tuples that only depend on the tuples before the barrier. Once a PE receives barriers from all it input links, it checkpoints its state, and then emits barrier and continues its computations.

1.3.3 Graph Data Processing Platforms

Graph is a well-known flexible abstraction for describing linked data, and a natural way of modeling a variety of problems across various domains. Although graph theory is well studied in mathematic, physics, and computer science over the years, the traditional graph algorithms often fail to provide a good performance when applied to big graphs. In fact, processing of large graphs that cannot fit in the memory of a single machine, brings about new challenges.

While the intuitive approach to overcome the size limitation, is to partition the data and parallelize the computation, data partitioning in a graph is not straightforward, because each vertex of a graph should be processed in the context of its surrounding vertices. Hence, the *data-parallelism* in systems like MapReduce and Spark, do not necessarily show a good performance for large-scale graphs. *Graph-parallel* processing model, is an alternative to data-parallel model, and has proven efficient and effective for large graph processing. In data-parallel computation, there is a *record-centric* view of data, and computation is done in parallel on separate and independent data records. On the other hand, in graph-parallel computation, a *vertex-centric* view of graphs is used, and the computation is done in parallel on all the vertices, each having access to its neighboring vertices.

In this section we present four different graph processing platforms, i.e., Pregel [27], GraphLab [2], PowerGraph [28], and GraphX [29].

Pregel

Pregel is a large-scale graph processing system, developed at Google, and inspired by the *bulk synchronous parallel (BSP)* model [30]. In the BSP model, there exists a set of processor-memory pairs that are communicating in a point-to-point manner, and there is a barrier mechanism to synchronize them. Giraph [31] is the open source counterpart of Pregel, developed as an Apache project.

Pregel executes an applications as a sequence of iterations, referred to as *supersteps*. In a superstep, a vertex receives all the messages sent to it in the previous superstep, updates its local state, and sends messages to its neighbors, to be delivered in the next superstep. Vertices use *message passing* to communicate directly with each other. A vertex can be either *active* or *inactive*. Initially, all the vertices are in the active state, but if they do not receive any message during a superstep they become inactive. Note that an inactive vertex becomes active again, as soon as it receives some messages in the subsequent supersteps. The algorithm terminates when all vertices are simultaneously inactive and there are no messages in transit.

Pregel uses the master-worker model, where the master coordinates workers, decides the number of partitions, and assigns partitions to workers. Each worker maintains the state of its partitions, executes the process of its vertices, and handles the message exchange with other workers. As mentioned earlier, graph partitioning is a crucial step in all the graph processing platforms that deal with huge graphs. Nevertheless, Pregel uses a naïve graph partitioning, by assigning vertices randomly to different machines. The random partitioning is expected to impose a high network traffic, because neighbors of a vertex are most likely not located on same machine (specially if the number of partitions is large) and thus, can not be accessed locally.

Fault tolerance in Pregel is achieved by checkpointing, meaning that master asks the workers to save their states at start of every k supersteps. This state includes the value/state of all the vertices and edges, as well as the incoming messages. If the master detects the failure of a worker, it tells all workers to revert to the last checkpoint, and resume the work from there.

GraphLab

Although Pregel makes large-scale graph-processing possible, it is limited in effect by its rigid synchronization mechanism. Considering the fact that the workload is not necessarily evenly distributed (due to the random partitioning) and taking into account the heterogeneous power of worker machines, the runtime of each superstep in Pregel is determined by the slowest machine in that superstep.

GraphLab utilizes an asynchronous model for graph processing. In this model vertices can read and modify the data in their *scope* directly, instead of sending read/update requests through messages passing. The scope of a vertex is the data stored in that vertex and in all its adjacent vertices and edges. All vertices, then, run in parallel, and the user defined function in each vertex has access to all the data in its scope. Note that vertex scopes are overlapping, meaning that vertices are shared among each other's scope. The overlapping scopes may cause a race condition when two update functions execute simultaneously on the same vertices.

To solve this problem, GraphLab defines three levels of consistency: (i) *full-consistency*, where during the execution of a function at vertex v, no other vertices can read or modify data within the scope of v, (ii) *edge consistency*, where during the execution of a function at vertex v, no other function can be applied to v and its adjacent edges, but the data in its adjacent vertices can be read, and (iii) *vertex consistency*, during the execution a function *v*, no other vertices can read or modify data at *v*. The stronger consistency level is used, the lower level of parallelization takes place. In the full consistency model, which is the strongest level, only vertices in the non-overlapping scopes can run in parallel, while in the vertex consistency level, all vertices can execute their functions in parallel.

To make GraphLab fault tolerant, two synchronous and asynchronous checkpointing models are proposed. In the synchronous model, the master periodically signals all workers to store their cached data, i.e., data that has been modified since the last checkpoint, to disk. The asynchronous model, however, is inspired by the Chandy-Lamport algorithm [26]. In this model, the checkpoint function is implemented as a function in all vertices with higher priority than all other functions, and the edge consistency model is used among them. The checkpoint function, then, is called periodically by each vertex to save its current vertex state, as well as the state of all the edges connected to not-checkpointed vertices.

GraphLab uses *two-phase partitioning* to split the input graph. In this model, the input graph is first turned into a smaller graph, called *meta-graph*, by grouping neighboring vertices and replacing them with a super node. Since the size of the meta-graph is much smaller than the original one, a fast balanced partition algorithm can be easily applied on it. When the meta-graph is divided into a number of partitions, each called an *atom*, the workers become responsible for one or more atoms each.

PowerGraph

PowerGraph improved on GraphLab, by: (i) introducing a new graph programming model, and (ii) employing a new *vertex-cut* partitioning. It factorizes the user defined functions in GraphLab into three steps of *gather, apply and scatter (GAS)*. In the gather step, a vertex accumulates data from its neighbors. Then, in the apply step, the user defined function is applied on the accumulated data and the vertex state is updated accordingly. Finally, in the scatter step, the vertex updates its adjacent edges and vertices. Initially all the vertices are active, and once a vertex function completes the scatter phase it becomes inactive. A vertex can become active again and then activate its neighboring vertices. The order in which active vertices are processed, is up to the PowerGraph execution engine.

Two synchronization modes can be used in PowerGraph. First is the synchronous mode, similar to Pregel, which uses the BSP model by defining supersteps. In each superstep, it executes the gather, apply, and scatter for all the active vertices with a barrier at the end. When all the workers complete their tasks, then, updates made to the vertices and edges are committed, and will be visible in the subsequent supersteps. Next mode is the asynchronous mode, in which changes made to vertices and edges during the apply and scatter functions are immediately committed to the graph, and are visible to the neighboring vertices.

The second big improvement of PowerGraph over GraphLab is replacing the *edge-cut* partitioning with *vertex-cut* partitioning. A vertex-cut partitioning divides edges of a graph into equal size clusters. The vertices that hold the endpoints of an edge are also placed in the same cluster as the edge itself. However, the vertices are not unique across clusters and might have to be replicated, due to the distribution of their edges across different clusters. A good vertex-cut is one that requires minimum number of replicas. Both theory and practice [32, 33] prove that power-law graphs can be efficiently processed in parallel, if vertex-cuts are used instead of edge-cuts, which is mainly due to unbalanced number of edges in each cluster in the edge-cut partitioning. PowerGraph takes this partitioning model into account and presents a new greedy algorithm for vertex-cut partitioning. The graph is read as a sequence of edges, and the master decides where to put the end point vertices of the received edge, based on their current membership in the existing partitions.

GraphX

GraphX is a graph processing platform, implemented on top of Spark, that unifies data-parallel and graph-parallel models. GraphX introduces the *property graph*, a new data structure and API that blurs the distinction between tables and graphs. In other words, the property graph makes it possible to express both table and graph views of the same physical data. Each table and graph view, then, has its own operators that exploit the semantics of the view to achieve efficient execution. This characteristic makes GraphX very efficient for running a pipeline of graph analytic tasks, where we have to switch between table and graph views frequently. The property graph is represented using two RDDs for vertices and edges, and an auxiliary table, which is a logical map from a vertex to the set of partitions that contain edges adjacent to that vertex. To partition the input graph, GraphX uses a vertex-cut partitioning, similar to PowerGraph.

1.3.4 Structured Data Processing Platforms

In the systems we have seen so far, data structure or schema is not considered. However, there are systems that are developed to exploit data schema in order to achieve an even better performance and ease of use. In this section, we introduce two of these systems, Hive [34] and Spark SQL [35].

Hive

Hive, initially developed at Facebook, is a system for managing and querying structured data. It is built on top of MapReduce and converts a query to a series of map and reduce tasks to run. Hive reuses the table data model in RDBMS, where a table is a set of rows with the same schema (columns). In Hive, each table corresponds to a HDFS directory. To work with tables, Hive uses HiveQL, a SQL-like query language that supports Data Definition Language (DDL) operations, e.g., `create`, `alter`, `drop`, as well as Data Manipulation Language (DML) operations, e.g., `load` and `insert` (overwrite), and also data retrieval query operations, e.g., `select`, `filter`, `join`, `group by`. It does not, however, support any operation for updating and deleting data items (rows).

To execute a query, Hive processes HiveQL statements and generates the execution plan in three phases: (i) parsing query that is to transform a query string to a parse tree representation, (ii) generating a logical plan from the parse tree representation, and optimizing the plan, and (iii) generating a physical plan by splitting the optimized logical plan into multiple map and reduce tasks.

Spark SQL

Hive conducts some optimizations in the logical plan generation to improve the performance, however Shark [36] pushes the performance improvement further, by replacing the MapReduce physical execution engine with Spark. More specifically, Shark is built on the Hive code base, but the physical execution engine part of Hive is replaced with Spark. Although Shark enables users to speed up their queries, the complicated code base that it inherits from Hive, brings about many challenges for query optimization. This is due to the fact that the optimization techniques used in Hive were designed for the MapReduce engine, not the Spark engine. Consequently, Spark SQL [35] was developed that borrowed data loading process from Hive, and in-memory column-oriented data store from Shark. Moreover, Spark SQL introduces some new features, for example, it enables adding schema to RDDs, and uses an RDD-aware optimizer, called catalyst optimizer.

Spark SQL introduces *DataFrame*, a distributed collection of rows with a homogeneous schema. DataFrame is equivalent to a table in a relational database, but it can also be manipulated in similar ways to RDDs. To have access to the functions of Spark SQL we need to build an `SQLContext`, just like we used an `SparkContext` as the entry point into Spark functionalities. By using an instance of `SQLContext`, one can build DataFrames from an existing RDD, from a Hive table, or from other data sources. Spark SQL provides a rich set of domain-specific languages for structured data manipulation with DataFrames.

Another feature added by Spark SQL is the catalyst optimizer, which is used in four phases: (i) to analyze the logical plan and resolve attribute references by tracking tables in data sources, (ii) to optimize the logical plan by applying standard optimization rules, e.g., null propagation, constant folding, boolean and filter simplifications, push predicate through joins and projection, etc., (iii) to generate several physical plans using Spark physical operators and to select a plan using some cost model, e.g., based on join algorithms, and finally (iv) to generate Java bytecode to run on workers.

1.4 Concluding Remarks

The unprecedented growth of data we have witnessed in recent years have brought about new challenges. This big data is commonly characterized by its extreme dimensions in terms of volume, the speed with which it is updated or produced, or the heterogeneity of its representation schemes. These properties have caused the traditional platforms fall short to store and process data efficiently, and thus, several new solutions are developed, among which we briefly explored a few of the state-of-the-art platforms. Each of these systems, of course, deserve more elaborated descriptions, but we kept it short, because our main goal was to position each system relative to other systems in the big data ecosystem. These systems and platforms are continuously evolving, but even if the tools and technologies for dealing with big data change over time, the main challenges and requirements that this chapter touched upon will remain the same. As opposed to many other technology hypes that go out of fashion in a few years, big data is here to stay.

Bibliography

[1] D. DeWitt and J. Gray, “Parallel database systems: the future of high performance database systems,” *Communications of the ACM*, vol. 35, no. 6, pp. 85–98, 1992.

[2] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, “Distributed graphlab: a framework for machine learning and data mining in the cloud,” *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[3] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, “Storm@ twitter,” in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 147–156.

[4] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[5] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[6] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas, “Apache flink: Stream and batch processing in a single engine,” *Data Engineering*, p. 28, 2015.

[7] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The google file system,” in *ACM SIGOPS operating systems review*, vol. 37, no. 5. ACM, 2003, pp. 29–43.

[8] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue, “Flat datacenter storage,” in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 1–15.

[9] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, “Ceph: A scalable, high-performance distributed file system,” in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 307–320.

[10] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. IEEE, 2010, pp. 1–10.

[11] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, 2011, pp. 1–7.

[12] Apache activemq. [Online]. Available: http://activemq.apache.org

[13] A. Richardson *et al.*, "Introduction to rabbitmq," *Google UK, Sep*, vol. 25, 2008.

[14] Apache flume. [Online]. Available: https://flume.apache.org

[15] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *USENIX Annual Technical Conference*, vol. 8, 2010, p. 9.

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 205–220, 2007.

[17] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[18] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.

[19] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah, "Randomized gossip algorithms," *IEEE/ACM Transactions on Networking (TON)*, vol. 14, no. SI, pp. 2508–2530, 2006.

[20] M. N. Vora, "Hadoop-hbase for large-scale data," in *Computer science and network technology (ICCSNT), 2011 international conference on*, vol. 1. IEEE, 2011, pp. 601–605.

[21] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 335–350.

[22] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, vol. 11, 2011, pp. 22–22.

[23] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, “Apache hadoop yarn: Yet another resource negotiator,” in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.

[24] H. C. Andrade, B. Gedik, and D. S. Turaga, *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. Cambridge University Press, 2014.

[25] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, “Discretized streams: Fault-tolerant streaming computation at scale,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 423–438.

[26] K. M. Chandy and L. Lamport, “Distributed snapshots: determining global states of distributed systems,” *ACM Transactions on Computer Systems (TOCS)*, vol. 3, no. 1, pp. 63–75, 1985.

[27] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: a system for large-scale graph processing,” in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[28] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, “Powergraph: Distributed graph-parallel computation on natural graphs,” in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 17–30.

[29] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, “Graphx: Graph processing in a distributed dataflow framework,” in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 599–613.

[30] L. G. Valiant, “A bridging model for parallel computation,” *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[31] C. Avery, “Giraph: Large-scale graph processing infrastructure on hadoop,” *Proceedings of the Hadoop Summit. Santa Clara*, vol. 11, 2011.

[32] A. Abou-Rjeili and G. Karypis, “Multilevel algorithms for partitioning power-law graphs,” in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2006, pp. 10–pp.

[33] K. Lang, “Finding good nearly balanced cuts in power law graphs,” *Preprint*, 2004.

[34] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, “Hive: a warehousing solution over a map-reduce framework,” *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.

[35] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, “Spark sql: Relational data processing in spark,” in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 1383–1394.

[36] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, “Shark: Sql and rich analytics at scale,” in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*. ACM, 2013, pp. 13–24.