1.1 Big Data Overview

Data is created constantly, and at an ever-increasing rate. Mobile phones, social media, imaging technologies to determine a medical diagnosis—all these and more create new data, and that must be stored somewhere for some purpose. Devices and sensors automatically generate diagnostic information that needs to be stored and processed in real time. Merely keeping up with this huge influx of data is difficult, but substantially more challenging is analyzing vast amounts of it, especially when it does not conform to traditional notions of data structure, to identify meaningful patterns and extract useful information. These challenges of the data deluge present the opportunity to transform business, government, science, and everyday life.

Several industries have led the way in developing their ability to gather and exploit data:

- Credit card companies monitor every purchase their customers make and can identify fraudulent purchases with a high degree of accuracy using rules derived by processing billions of transactions.
- Mobile phone companies analyze subscribers' calling patterns to determine, for example, whether a caller's frequent contacts are on a rival network. If that rival network is offering an attractive promotion that might cause the subscriber to defect, the mobile phone company can proactively offer the subscriber an incentive to remain in her contract.
- For companies such as LinkedIn and Facebook, data itself is their primary product. The valuations of these companies are heavily derived from the data they gather and host, which contains more and more intrinsic value as the data grows.

Three attributes stand out as defining Big Data characteristics:

- **Huge volume of data:** Rather than thousands or millions of rows, Big Data can be billions of rows and millions of columns.
- **Complexity of data types and structures:** Big Data reflects the variety of new data sources, formats, and structures, including digital traces being left on the web and other digital repositories for subsequent analysis.
- **Speed of new data creation and growth:** Big Data can describe high velocity data, with rapid data ingestion and near real time analysis.

Although the volume of Big Data tends to attract the most attention, generally the variety and velocity of the data provide a more apt definition of Big Data. (Big Data is sometimes described as having 3 Vs: volume, variety, and velocity.) Due to its size or structure, Big Data cannot be efficiently analyzed using only traditional databases or methods. Big Data problems require new tools and technologies to store, manage, and realize the business benefit. These new tools and technologies enable creation, manipulation, and management of large datasets and the storage environments that house them. Another definition of Big Data comes from the McKinsey Global report from 2011:**Big Data is data whose scale,**

distribution, diversity, and/or timeliness require the use of new technical architectures and analytics to enable insights that unlock new sources of business value.

McKinsey & Co.; Big Data: The Next Frontier for Innovation, Competition, and Productivity [1]

McKinsey's definition of Big Data implies that organizations will need new data architectures and analytic sandboxes, new tools, new analytical methods, and an integration of multiple skills into the new role of the data scientist, which will be discussed in Section 1.3. Figure 1.1 highlights several sources of the Big Data deluge.

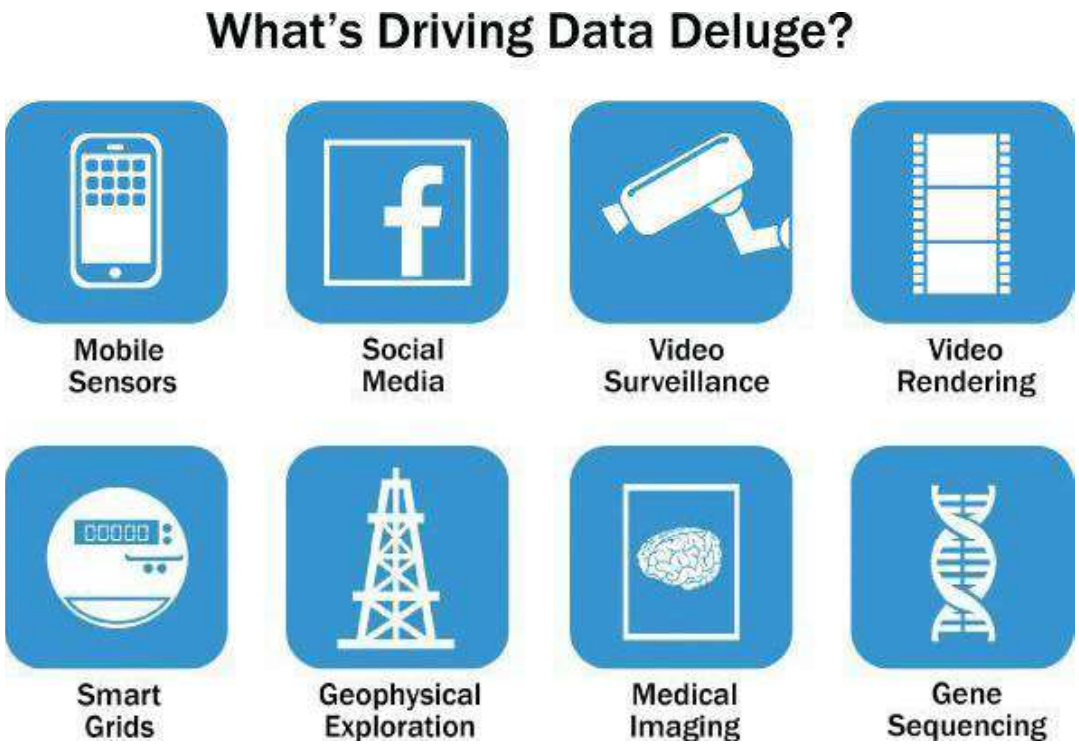


Figure 1.1 What's driving the data deluge

The rate of data creation is accelerating, driven by many of the items in Figure 1.1.

Social media and genetic sequencing are among the fastest-growing sources of Big Data and examples of untraditional sources of data being used for analysis.

For example, in 2012 Facebook users posted 700 status updates per second worldwide, which can be leveraged to deduce latent interests or political views of users and show relevant ads. For instance, an update in which a woman changes her relationship status from "single" to "engaged" would trigger ads on bridal dresses, wedding planning, or name-changing services.

Facebook can also construct social graphs to analyze which users are connected to each other as an interconnected network. In March 2013, Facebook released a new feature called "Graph Search," enabling users and developers to search social graphs for people with similar interests, hobbies, and shared locations.

Another example comes from genomics. Genetic sequencing and human genome mapping provide a detailed understanding of genetic makeup and lineage. The health care industry is looking toward these advances to help predict which illnesses a person is likely to get in his lifetime and take steps to avoid these maladies or reduce their impact through the use

of personalized medicine and treatment. Such tests also highlight typical responses to different medications and pharmaceutical drugs, heightening risk awareness of specific drug treatments.

While data has grown, the cost to perform this work has fallen dramatically. The cost to sequence one human genome has fallen from $100 million in 2001 to $10,000 in 2011, and the cost continues to drop. Now, websites such as 23andme (Figure 1.2) offer genotyping for less than $100. Although genotyping analyzes only a fraction of a genome and does not provide as much granularity as genetic sequencing, it does point to the fact that data and complex analysis is becoming more prevalent and less expensive to deploy.

Figure 1.2 Examples of what can be learned through genotyping, from 23andme.com

As illustrated by the examples of social media and genetic sequencing, individuals and organizations both derive benefits from analysis of ever-larger and more complex datasets that require increasingly powerful analytical capabilities.

1.1.1 Data Structures

Big data can come in multiple forms, including structured and non-structured data such as financial data, text files, multimedia files, and genetic mappings. Contrary to much of the traditional data analysis performed by organizations, most of the Big Data is unstructured or semi-structured in nature, which requires different techniques and tools to process and analyze. [2] Distributed computing environments and massively parallel processing (MPP) architectures that enable parallelized data ingest and analysis are the preferred approach to process such complex data.

With this in mind, this section takes a closer look at data structures.

Figure 1.3 shows four types of data structures, with 80–90% of future data growth coming from non-structured data types. [2] Though different, the four are commonly mixed. For example, a classic Relational Database Management System (RDBMS) may store call logs for a software support call center. The RDBMS may store characteristics of the support calls as typical structured data, with attributes such as time stamps, machine type, problem type, and operating system. In addition, the system will likely have unstructured, quasi- or semi-structured data, such as free-form call log information taken from an e-mail ticket of the problem, customer chat history, or transcript of a phone call describing the technical problem and the solution or audio file of the phone call conversation. Many insights could be extracted from the unstructured, quasi- or semi-structured data in the call center data.

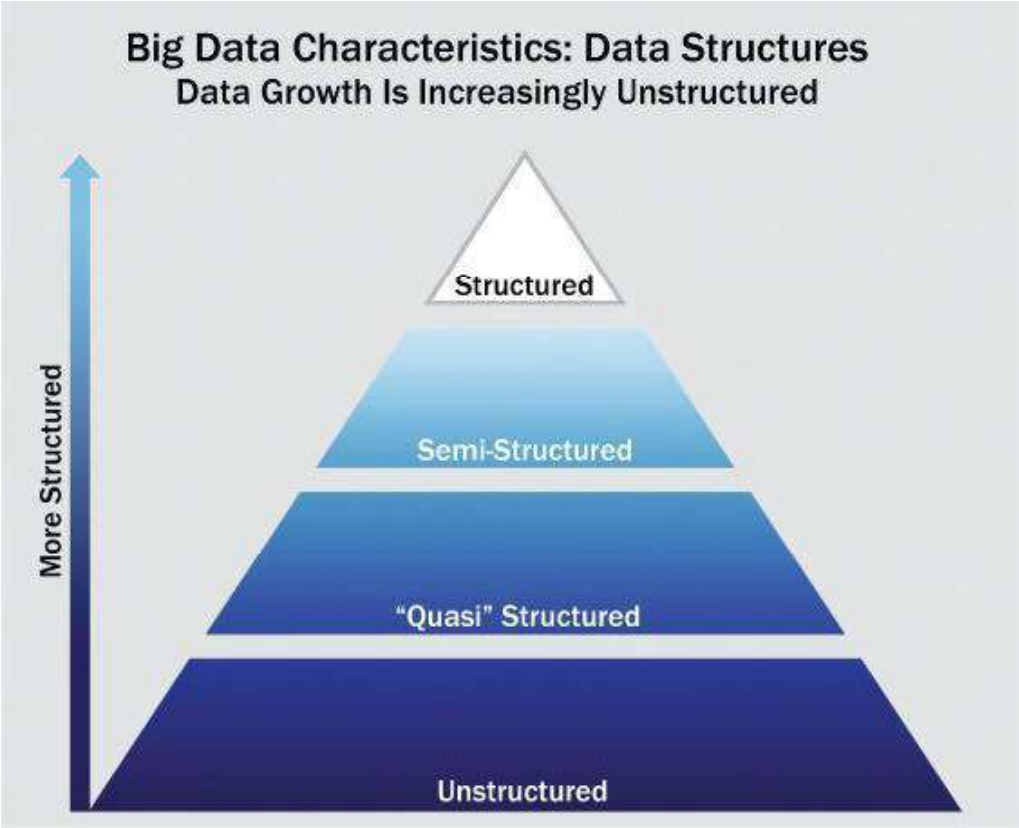


Figure 1.3 Big Data Growth is increasingly unstructured

Although analyzing structured data tends to be the most familiar technique, a different technique is required to meet the challenges to analyze semi-structured data (shown as XML), quasi-structured (shown as a clickstream), and unstructured data.

Here are examples of how each of the four main types of data structures may look.

- **Structured data:** Data containing a defined data type, format, and structure (that is, transaction data, online analytical processing [OLAP] data cubes, traditional RDBMS, CSV files, and even simple spreadsheets). See Figure 1.4.
- **Semi-structured data:** Textual data files with a discernible pattern that enables parsing (such as Extensible Markup Language [XML] data files that are self-describing and defined by an XML schema). See Figure 1.5.
- **Quasi-structured data:** Textual data with erratic data formats that can be formatted with effort, tools, and time (for instance, web clickstream data that may contain inconsistencies in data values and formats). See Figure 1.6.
- **Unstructured data:** Data that has no inherent structure, which may include text documents, PDFs, images, and video. See Figure 1.7.

SUMMER FOOD SERVICE PROGRAM 1]

(Data as of August 01, 2011)

Fiscal Year	Number of Sites	Peak (July) Participation	Meals Served	Total Federal Expenditures 2]
	----------Thousands----------		--Mil.--	---Million $---
1969	1.2	99	2.2	0.3
1970	1.9	227	8.2	1.8
1971	3.2	569	29.0	8.2
1972	6.5	1,080	73.5	21.9
1973	11.2	1,437	65.4	26.6
1974	10.6	1,403	63.6	33.6
1975	12.0	1,785	84.3	50.3
1976	16.0	2,453	104.8	73.4
TQ 3]	22.4	3,455	198.0	88.9
1977	23.7	2,791	170.4	114.4
1978	22.4	2,333	120.3	100.3
1979	23.0	2,126	121.8	108.6
1980	21.6	1,922	108.2	110.1
1981	20.6	1,726	90.3	105.9
1982	14.4	1,397	68.2	87.1
1983	14.9	1,401	71.3	93.4
1984	15.1	1,422	73.8	96.2
1985	16.0	1,462	77.2	111.5
1986	16.1	1,509	77.1	114.7
1987	16.9	1,560	79.9	129.3
1988	17.2	1,577	80.3	133.3
1989	18.5	1,652	86.0	143.8
1990	19.2	1,692	91.2	163.3

Figure 1.4 Example of structured data

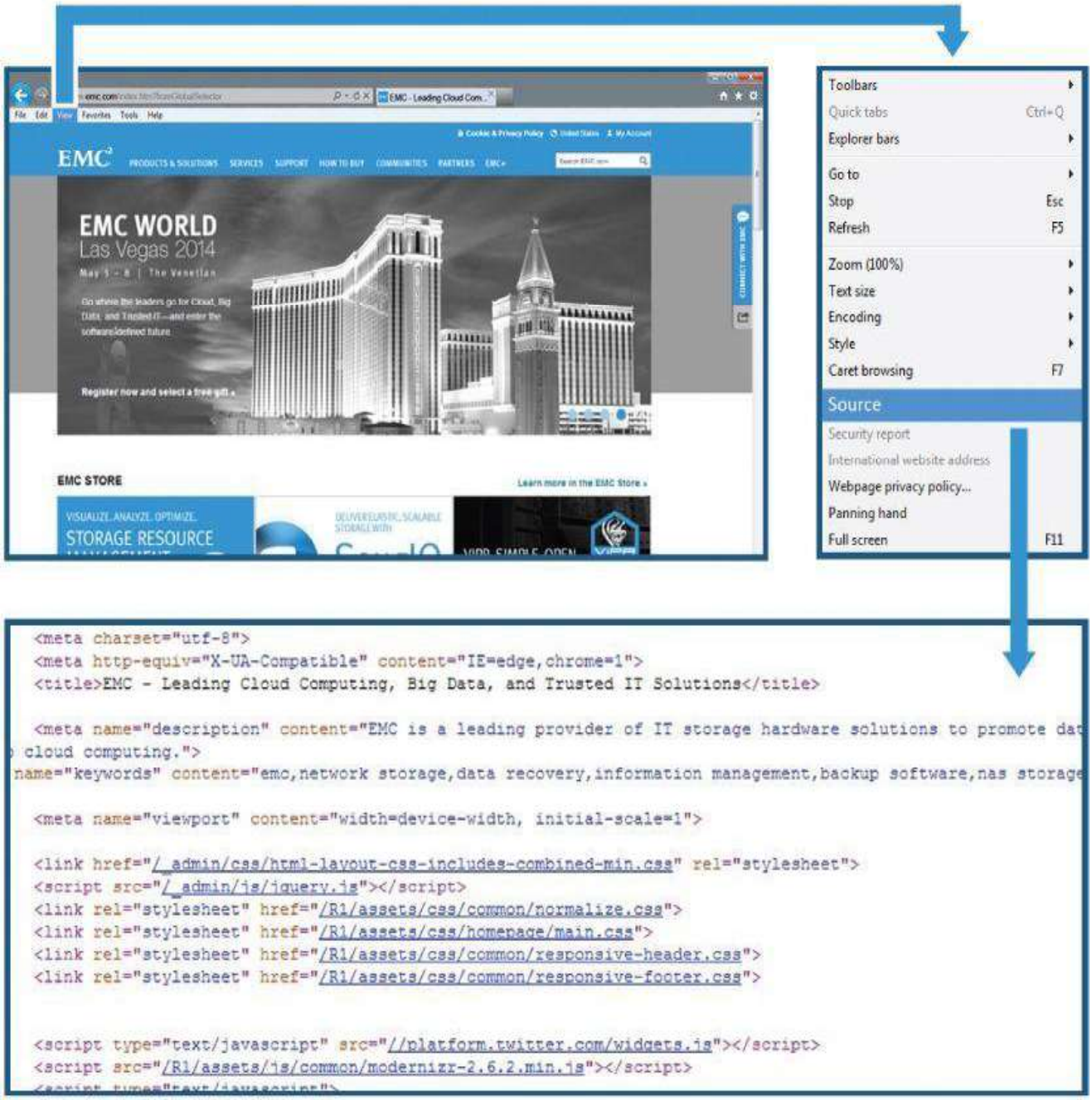


Figure 1.5 Example of semi-structured data

Figure 1.6 Example of EMC Data Science search results

Figure 1.7 Example of unstructured data: video about Antarctica expedition [3]

Quasi-structured data is a common phenomenon that bears closer scrutiny. Consider the following example. A user attends the EMC World conference and subsequently runs a Google search online to find information related to EMC and Data Science. This would produce a URL such as `https://www.google.com/#q=EMC+ data+science` and a list of results, such as in the first graphic of Figure 1.5.

After doing this search, the user may choose the second link, to read more about the headline "Data Scientist—EMC Education, Training, and Certification." This brings the user to an `emc.com` site focused on this topic and a new URL, `https://education.emc.com/guest/campaign/data science.aspx`, that displays the page shown as (2) in Figure 1.6. Arriving at this site, the user may decide to click to learn more about the process of becoming certified in data science. The user chooses a link toward the top of the page on Certifications, bringing the user to a new URL: `https://education.emc.com/guest/certification/framework/stf/data science.asp` which is (3) in Figure 1.6.

Visiting these three websites adds three URLs to the log files monitoring the user's computer or network use. These three URLs are:

`https://www.google.com/#q=EMC+data+science`

`https://education.emc.com/guest/campaign/data science.aspx`

`https://education.emc.com/guest/certification/framework/stf/data science.asp`

This set of three URLs reflects the websites and actions taken to find Data Science information related to EMC. Together, this comprises a *clickstream* that can be parsed and mined by data scientists to discover usage patterns and uncover relationships among clicks and areas of interest on a website or group of sites.

The four data types described in this chapter are sometimes generalized into two groups:

structured and unstructured data. Big Data describes new kinds of data with which most organizations may not be used to working. With this in mind, the next section discusses common technology architectures from the standpoint of someone wanting to analyze Big Data.

1.1.2 Analyst Perspective on Data Repositories

The introduction of spreadsheets enabled business users to create simple logic on data structured in rows and columns and create their own analyses of business problems. Database administrator training is not required to create spreadsheets: They can be set up to do many things quickly and independently of information technology (IT) groups. Spreadsheets are easy to share, and end users have control over the logic involved. However, their proliferation can result in "many versions of the truth." In other words, it can be challenging to determine if a particular user has the most relevant version of a spreadsheet, with the most current data and logic in it. Moreover, if a laptop is lost or a file becomes corrupted, the data and logic within the spreadsheet could be lost. This is an ongoing challenge because spreadsheet programs such as Microsoft Excel still run on many computers worldwide. With the proliferation of data islands (or spreadmarts), the need to centralize the data is more pressing than ever.

As data needs grew, so did more scalable data warehousing solutions. These technologies enabled data to be managed centrally, providing benefits of security, failover, and a single repository where users could rely on getting an "official" source of data for financial reporting or other mission-critical tasks. This structure also enabled the creation of OLAP cubes and BI analytical tools, which provided quick access to a set of dimensions within an RDBMS. More advanced features enabled performance of in-depth analytical techniques such as regressions and neural networks. Enterprise Data Warehouses (EDWs) are critical for reporting and BI tasks and solve many of the problems that proliferating spreadsheets introduce, such as which of multiple versions of a spreadsheet is correct. EDWs—and a good BI strategy—provide direct data feeds from sources that are centrally managed, backed up, and secured.

Despite the benefits of EDWs and BI, these systems tend to restrict the flexibility needed to perform robust or exploratory data analysis. With the EDW model, data is managed and controlled by IT groups and database administrators (DBAs), and data analysts must depend on IT for access and changes to the data schemas. This imposes longer lead times for analysts to get data; most of the time is spent waiting for approvals rather than starting meaningful work. Additionally, many times the EDW rules restrict analysts from building datasets. Consequently, it is common for additional systems to emerge containing critical data for constructing analytic datasets, managed locally by power users. IT groups generally dislike existence of data sources outside of their control because, unlike an EDW, these datasets are not managed, secured, or backed up. From an analyst perspective, EDW and BI solve problems related to data accuracy and availability. However, EDW and BI introduce new problems related to flexibility and agility, which were less pronounced when dealing with spreadsheets.

A solution to this problem is the analytic sandbox, which attempts to resolve the conflict for analysts and data scientists with EDW and more formally managed corporate data. In this model, the IT group may still manage the analytic sandboxes, but they will be

2
Overview of Big Data Analytics

In this chapter, we will talk about big data analytics, starting with a general point of view and then taking a deep dive into some common technologies used to gain insights into data. This chapter introduces the reader to the process of examining large data sets to uncover patterns in data, generating reports, and gathering valuable insights. We will particularly focus on the seven Vs of big data. We will also learn about data analysis and big data; we will see the challenges that big data provides and how they are dealt with in distributed computing, and look at approaches using Hive and Tableau to showcase the most commonly used technologies.

In a nutshell, the following topics will be covered throughout this chapter:

- Introduction to data analytics
- Introduction to big data
- Distributed computing using Apache Hadoop
- MapReduce framework
- Hive
- Apache Spark

Introduction to data analytics

Data analytics is the process of applying qualitative and quantitative techniques when examining data, with the goal of providing valuable insights. Using various techniques and concepts, data analytics can provide the means to explore the data **exploratory data analysis** (**EDA**) as well as draw conclusions about the data **confirmatory data analysis** (**CDA**). The EDA and CDA are fundamental concepts of data analytics, and it is important to understand the differences between the two.

EDA involves the methodologies, tools, and techniques used to explore data with the intention of finding patterns in the data and relationships between various elements of the data. CDA involves the methodologies, tools, and techniques used to provide an insight or conclusion for a specific question, based on hypothesis and statistical techniques, or simple observation of the data.

Inside the data analytics process

Once data is deemed ready, it can be analyzed and explored by data scientists using statistical methods such as SAS. Data governance also becomes a factor to ensure the proper collection and protection of the data. Another less well known role is that of a **data steward** who specializes in understanding the data to the byte; exactly where it is coming from, all transformations that occur, and what the business really needs from the column or field of data.

Various entities in the business might be dealing with addresses differently, such as the following:

```
123 N Main St vs 123 North Main Street.
```

But, our analytics depend on getting the correct address field, else both the addresses mentioned will be considered different and our analytics will not have the same accuracy.

The analytics process starts with data collection based on what the analysts might need from the data warehouse, collecting all sorts of data in the organization (sales, marketing, employee, payroll, HR, and so on). Data stewards and governance teams are important here to make sure the right data is collected and that any information deemed confidential or private is not accidentally exported out, even if the end users are all employees. **Social Security Numbers** (**SSNs**) or full addresses might not be a good idea to include in analytics as this can cause a lot of problems to the organization.

Data quality processes must be established to make sure that the data being collected and engineered is correct and will match the needs of the data scientists. At this stage, the main goal is to find and fix data quality problems that could affect the accuracy of analytical needs. Common techniques are profiling the data, cleansing the data to make sure that the information in a dataset is consistent, and also that any errors and duplicate records are removed.

Analytical applications can thus be realized using several disciplines, teams, and skillsets. Analytical applications can be used to generate reports all the way to automatically triggering business actions. For example, you can simply a create daily sales report to be emailed out to all managers every day at 8 AM in the morning. But, you can also integrate with business process management applications or some custom stock trading applications to take action, such as buying, selling, or alerting on activities in the stock market. You can also think of taking in news articles or social media information to further influence what decisions to be made.

Data visualization is an important piece of data analytics and it's hard to understand numbers when you are looking at a lot of metrics and calculation. Rather, there is an increasing dependence on **business intelligence** (**BI**) tools, such as Tableau, QlikView, and so on, to explore and analyze the data. Of course, large-scale visualization, such as showing all Uber cars in the country or heat maps showing water supply in New York City, requires more custom applications or specialized tools to be built.

Managing and analyzing data has always been a challenge across many organizations of different sizes across all industries. Businesses have always struggled to find a pragmatic approach to capturing information about their customers, products, and services. When the company only had a handful of customers who bought a few of their items, it was not that difficult. It was not as big of a challenge. But over time, companies in the markets started growing. Things have become more complicated. Now, we have branding information and social media. We have things that are sold and bought over the internet. We need to come up with different solutions. With web development, organizations, pricing, social networks, and segmentations, there's a lot of different data that we're dealing with that brings a lot more complexity when it comes to dealing, managing, organizing, and trying to gain some insight from the data.

Introduction to big data

Twitter, Facebook, Amazon, Verizon, Macy's, and Whole Foods are all companies that run their business using data analytics and base many of the decisions on the analytics. Think about what kind of data they are collecting, how much data they might be collecting, and then how they might be using the data.

Let's look at the grocery store example seen earlier; what if the store starts expanding its business to set up hundreds of stores? Naturally, the sales transactions will have to be collected and stored at a scale hundreds of times more than the single store. But then, no business works independently any more. There is a lot of information out there, starting from local news, tweets, Yelp reviews, customer complaints, survey activities, competition from other stores, the changing demographics or economy of the local area, and so on. All such additional data can help in better understanding the customer behavior and the revenue models.

For example, if we see increasing negative sentiment regarding the store's parking facility, then we could analyze this and take corrective action such as validated parking or negotiating with the city's public transportation department to provide more frequent trains or buses for better reach. Such an increasing quantity and variety of data, while it provides better analytics also poses challenges to the business IT organization trying to store and process and analyze all the data. It is, in fact, not uncommon to see TBs of data.

Every day, we create more than two quintillion bytes of data (2 EB), and it is estimated that more than 90% of the data has been generated in last few years alone:

```
1 KB = 1024 Bytes
1 MB = 1024 KB
1 GB = 1024 MB
1 TB = 1024 GB ~ 1,000,000 MB
1 PB = 1024 TB ~ 1,000,000 GB ~ 1,000,000,000 MB
1 EB = 1024 PB ~ 1,000,000 TB ~ 1,000,000,000 GB ~ 1,000,000,000,000 MB
```

Such large amounts of data since the 1990s and the need to understand and make sense of the data, gave rise to the term big data.

In 2001, *Doug Laney*, then an analyst at consultancy Meta Group Inc (which got acquired by Gartner), introduced the idea of three Vs (that is, Variety, Velocity, and Volume). Now, we refer to four Vs instead of three Vs with the addition of Veracity of data to the three Vs.

The following are the four Vs of big data, used to describe the properties of big data.

Variety of data

Data can be obtained from a number of sources, such as weather sensors, car sensors, census data, Facebook updates, tweets, transactions, sales, and marketing. The data format is both structured and unstructured as well. Data types can also be different, binary, text, JSON, and XML. Variety really begins to scratch the surface of the depth of the data.

Velocity of data

Data can be from a data warehouse, batch mode file archives, near real-time updates, or instantaneous real-time updates from the Uber ride you just booked. Velocity refers to the increasing speed at which this data is created, and the increasing speed at which the data can be processed, stored, and analyzed by relational databases.

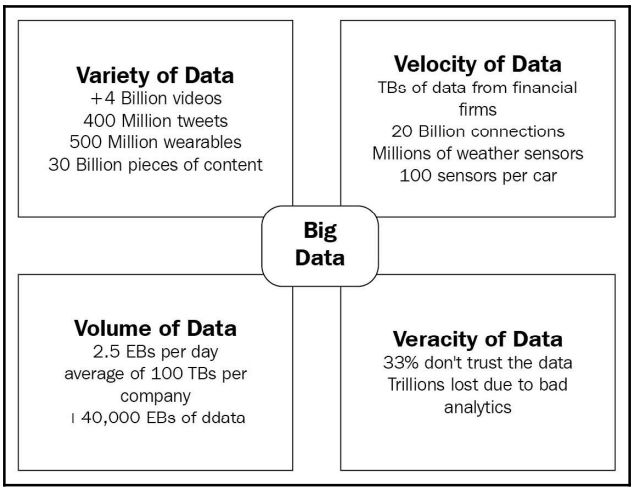
Volume of data

Data can be collected and stored for an hour, a day, a month, a year, or 10 years. The size of data is growing to 100s of TBs for many companies. Volume refers to the scale of the data, which is part of what makes big data big.

Veracity of data

Data can be analyzed for actionable insights, but with so much data of all types being analyzed from across data sources, it is very difficult to ensure correctness and proof of accuracy.

The following are the four Vs of big data:



To make sense of all the data and apply data analytics to big data, we need to expand the concept of data analytics to operate at a much larger scale that deals with the four Vs of big data. This changes not only the tools and technologies and methodologies used in analyzing the data, but also changes the way we even approach the problem. If a SQL database was used for data in a business in 1999, to handle the data now for the same business, we will need a distributed SQL database that is scalable and adaptable to the nuances of the big data space.

The four Vs described earlier are no longer sufficient to cover the capabilities and needs of big data analytics, hence nowadays it's common to hear about the seven Vs instead of the four Vs.

Variability of data

Variability refers to data whose meaning is constantly changing. Many a time, organizations need to develop sophisticated programs in order to be able to understand context in them and decode their exact meaning.

Visualization

Visualization comes in the picture when you need to present the data in a readable and accessible manner after it has been processed.

Value

Big data is large and is increasing everyday, however the data is also messy, noisy, and constantly changing. It is available for all in a variety of formats and is in no position to be used without analysis and visualization.

What is big data mining?

Big data mining forms the first of two broad categories of big data analytics, the other being Predictive Analytics, which we will cover in later chapters. In simple terms, big data mining refers to the entire life cycle of processing large-scale datasets, from procurement to implementation of the respective tools to analyze them.

The next few chapters will illustrate some of the high-level characteristics of any big data project that is undertaken in an organization.

Big data mining in the enterprise

Implementing a big data solution in a medium to large size enterprise can be a challenging task due to the extremely dynamic and diverse range of considerations, not the least of which is determining what specific business objectives the solution will address.

Building the case for a Big Data strategy

Perhaps the most important aspect of big data mining is determining the appropriate use cases and needs that the platform would address. The success of any big data platform depends largely on finding relevant problems in business units that will deliver measurable value for the department or organization. The hardware and software stack for a solution that collects large volumes of sensor or streaming data will be materially different from one that is used to analyze large volumes of internal data.

The following are some suggested steps that, in my experience, have been found to be particularly effective in building and implementing a corporate big data strategy:

- **Who needs big data mining**: Determining which business groups will benefit most significantly from a big data mining solution is the first step in this process. This would typically entail groups that are already working with large datasets, are important to the business, and have a direct revenue impact, and optimizing their processes in terms of data access or time to analyze information would have an impact on the daily work processes.
  As an example, in a pharmaceutical organization, this could include Commercial Research, Epidemiology, Health Economics, and Outcomes. At a financial services organization, this could include Algorithmic Trading Desks, Quantitative Research, and even Back Office.

- **Determining the use cases**: The departments identified in the preceding step might already have a platform that delivers the needs of the group satisfactorily. Prioritizing among multiple use cases and departments (or a collection of them) requires personal familiarity with the work being done by the respective business groups.
  Most organizations follow a hierarchical structure where the interaction among business colleagues is likely to be mainly along **rank lines**. Determining impactful analytics use cases requires a close collaboration between both the practitioner as well as the stakeholder; namely, both the management who has oversight of a department as well as the staff members who perform the hands-on analysis. The business stakeholder can shed light on which aspects of his or her business will benefit the most from more efficient data mining and analytics environment. The practitioners provide insight on the challenges that exist at the hands-on operational level. Incremental improvements that consolidate both the operational as well as the managerial aspects to determine an optimal outcome are bound to deliver faster and better results.

- **Stakeholders' buy-in**: The buy-in of the stakeholders—in other words, a consensus among decision-makers and those who can make independent budget decisions—should be established prior to commencing work on the use case(s). In general, multiple buy-ins should be secured for redundancy such that there is a pool of primary and secondary sources that can provide appropriate support and funding for an extension of any early-win into a broader goal. The buy-in process does not have to be deterministic and this may not be possible in most circumstances. Rather, a general agreement on the value that a certain use case will bring is helpful in establishing a baseline that can be leveraged on the successful execution of the use case.
- **Early-wins and the effort-to-reward ratio**: Once the appropriate use cases have been identified, finding the ones that have an optimal effort-to-reward ratio is critical. A relatively small use case that can be implemented in a short time within a smaller budget to optimize a specific business-critical function helps in showcasing early-wins, thus adding credibility to the big data solution in question. We cannot precisely quantify these intangible properties, but we can hypothesize:

$$\text{E-R Ratio} = \frac{\text{Time} + \text{Cost} + \text{Number of Resources} + \text{Criticality of Use Case}}{\textit{Business Value}}$$

In this case, *effort* is the time and work required to implement the use case. This includes aspects such as how long it would take to procure the relevant hardware and/or software that is part of the solution, the resources or equivalent *man-hours* it will take to implement the solution, and the overall operational overhead. An open source tool might have a lower barrier to entry relative to implementing a commercial solution that may involve lengthy procurement and risk analysis by the organization. Similarly, a project that spans across departments and would require time from multiple resources who are already engaged in other projects is likely to have a longer duration than one that can be executed by the staff of a single department. If the net effort is low enough, one can also run more than one exercise in parallel as long as it doesn't compromise the quality of the projects.

- **Leveraging the early-wins**: The successful implementation of one or more of the projects in the early-wins phase often lays the groundwork to develop a bigger strategy for the big data analytics platform that goes far beyond the needs of just a single department and has a broader organizational-level impact. As such, the early-win serves as a first, but crucial, step in establishing the value of big data to an audience, who may or may not be skeptical of its viability and relevance.

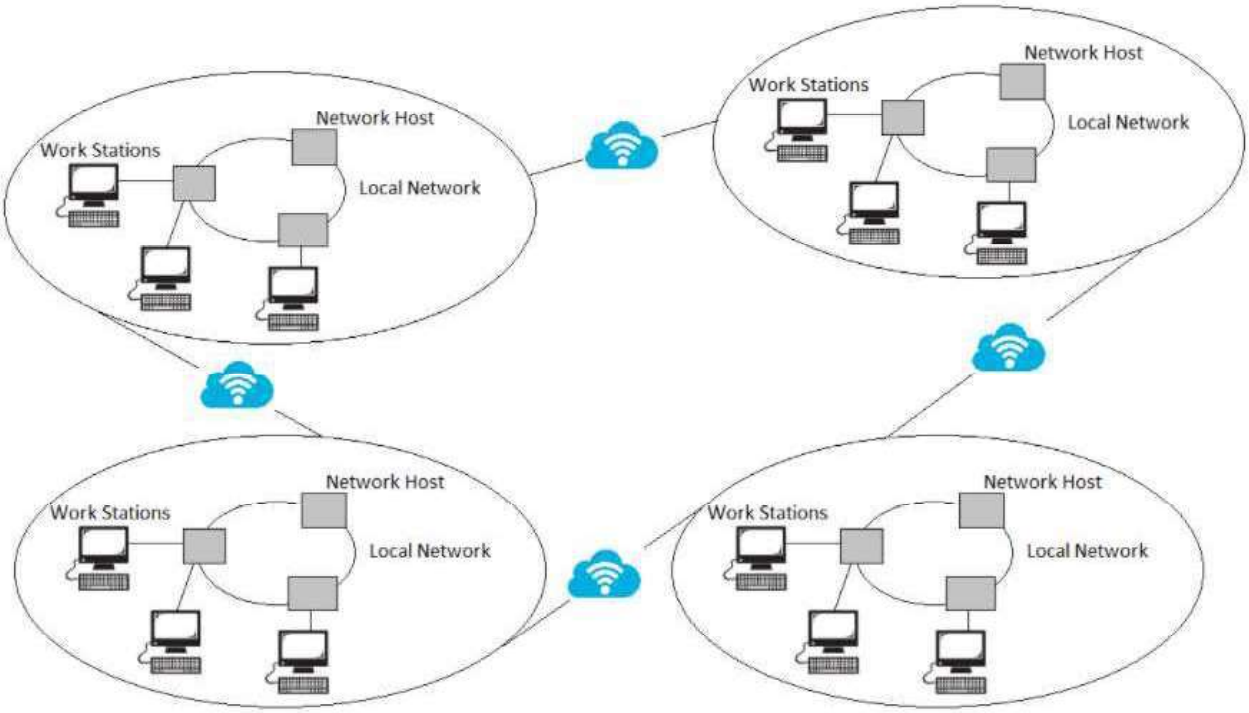
Implementation life cycle

As outlined earlier, the implementation process can span multiple steps. These steps are often iterative in nature and require a trial-and-error approach. This will require a fair amount of perseverance and persistence as most undertakings will be characterized by varying degrees of successes and failures.

In practice, a Big Data strategy will include multiple stakeholders and a collaborative approach often yields the best results. Business sponsors, business support and IT & Analytics are three broad categories of stakeholders that together create a proper unified solution, catering to the needs of the business to the extent that budget and IT capabilities will permit.

Distributed Computing for Big Data

- Usage of more than one connected computer
- Examples: NoW(Network of Workstations)



Distributed Computing for Big Data

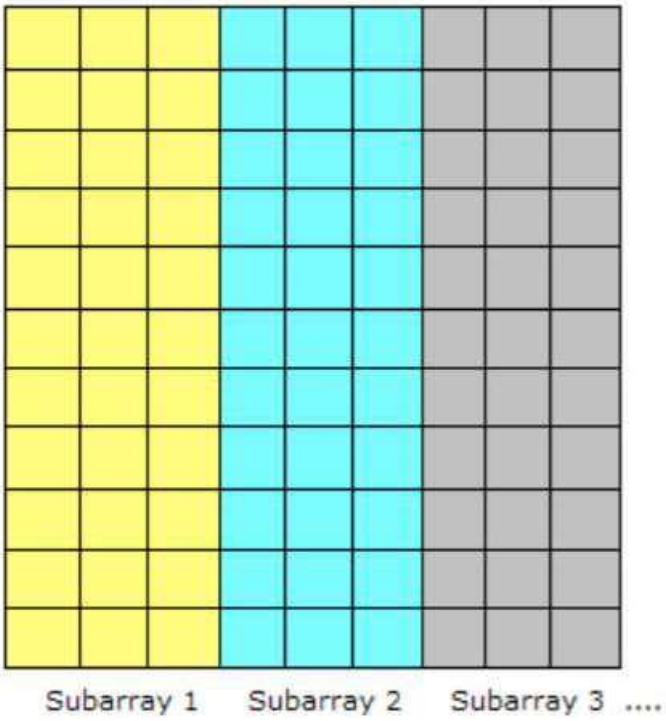
- Treated infrastructure as one big pool of computing, storage and networking resources
- Moves to another resource in the pool in case of failure
- **Fault Tolerance or High Availability**
- Cost of computing and storage resources has decreased considerably
- E.g Distributed File System

Parallel Computing for Big Data

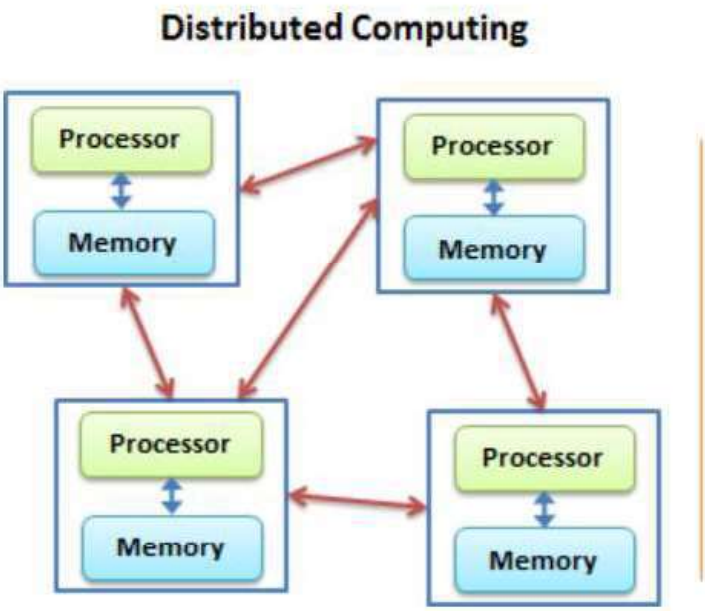
- Serial – Sequence of instructions one by one
- Start to Finish on a single processor
- Parallel Programming
  - Processing broken into multiple parts
  - Each part executed concurrently
  - Different CPUs on a single or different machines

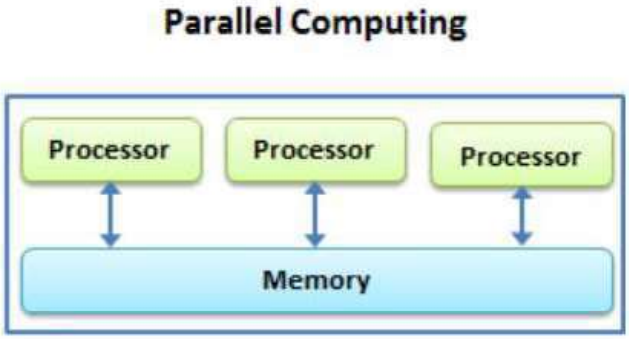
Parallel Computing for Big Data

- First step
  - Identify set of tasks that could be run concurrently
  - Data partitioned without dependencies



Distributed, Parallel Computing

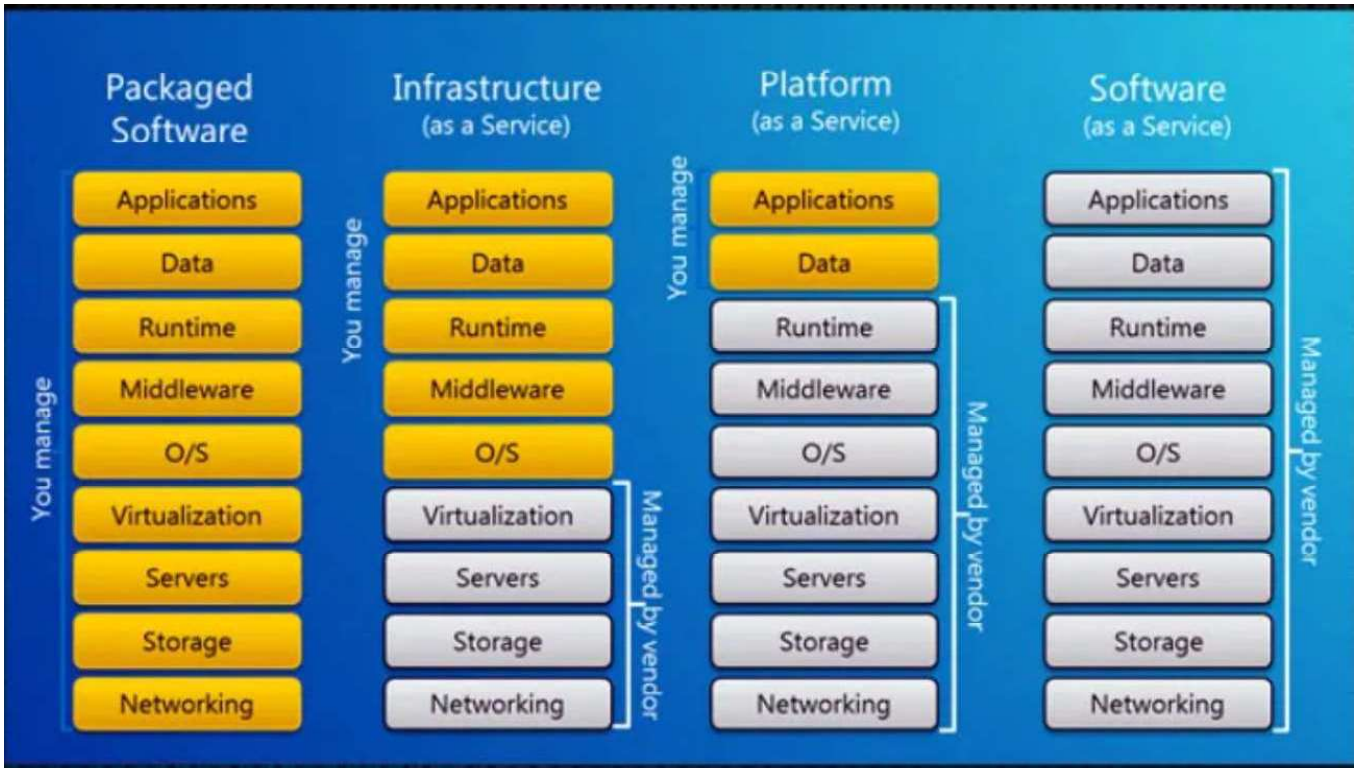




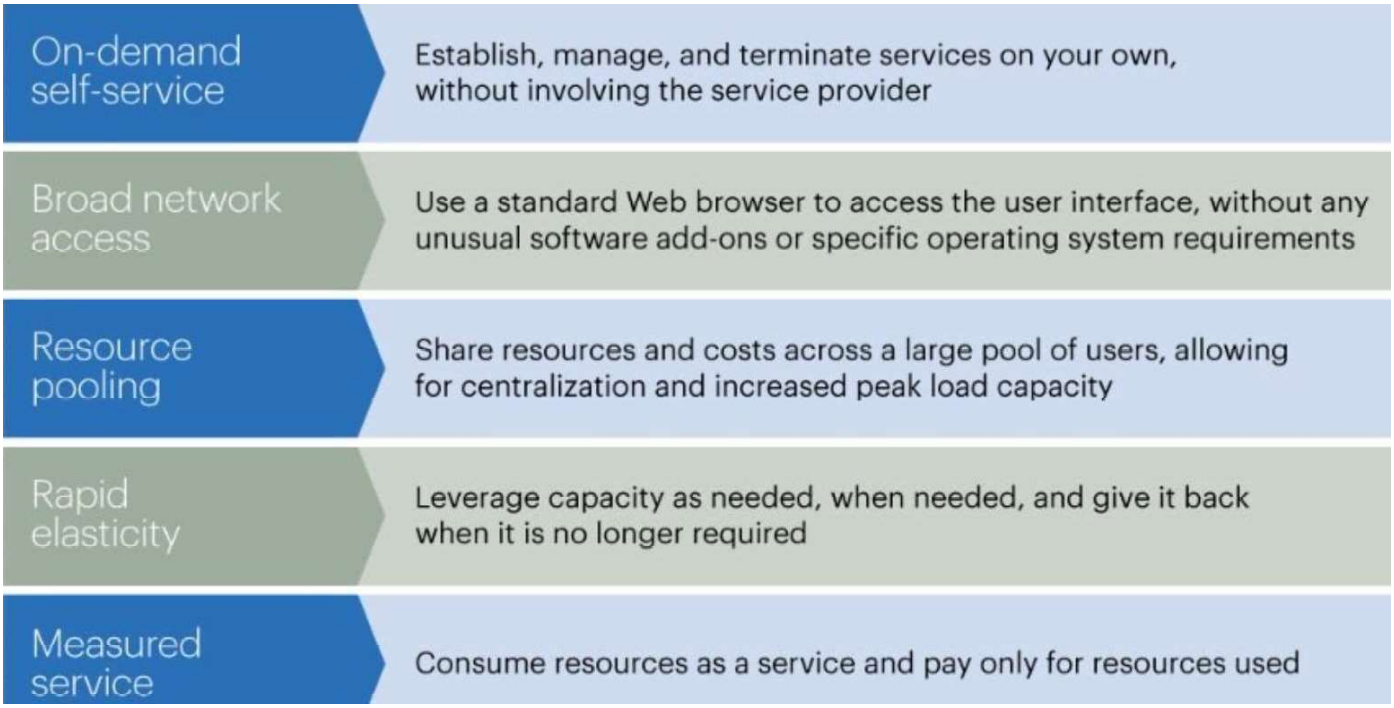
Cloud Computing for big data

- Set of high powered servers
- Pay as you use
- Memory or Storage intensive
- View and query large data sets quickly
- Infrastructure as a Service
- Platform as a Service
- Software as a Service

Cloud Computing for big data



Drivers of Cloud Computing for big data



Cloud Computing for big data

- Need not ship terabytes of data around
- Store in cloud and **move analytics close to data**
- Store insensitive data in cloud
- Sensitive data – Private Cloud/Clusters
- Advantages
  - Scalability
  - Cost reduction
  - Faster time to market

Cloud Computing for big data

- Limitations
  - Latency
  - Multi-tenancy overhead
  - Data privacy
  - Data Security
  - Costs are charged for static storage

In-memory computing for big data

- Computations are **latency sensitive**
- Have to be done **in memory**
- Disk reads and writes would cause delays
- For compute-intensive applications
  - Real time stream processing
  - Apache Spark

Big Data Mining

- Two broad categories of BDA
  - Big Data Mining
  - Predictive Analytics
- Big Data Mining
  - Refers to the entire life cycle of processing large data sets
  - From procurement of data to implementation of respective tools to analyze data

Big Data Mining – Building Corporate Big Data strategy

- Determine use cases
- Then fix the platforms

Steps

1. Who needs big data mining?
   - Which business groups will benefit significantly?
   - Revenue impact
   - E.g Pharmaceutical company – Commercial Research, Epidemiology, Health Economics

Big Data Mining – Building Corporate Big Data strategy

2. Determining use cases
   - Impactful analysis by working closer with practitioner (analyst) and stakeholder (business end user)
   - Analyst will work on operational challenges that might be faced in implementing the BD solution
   - Business user will highlight the aspects of business that will benefit from the BD solution
   - An optimal outcome consolidating both

Big Data Mining – Building Corporate Big Data strategy

3. Stakeholder's buy in
   - Budget decisions
   - Establish a baseline
   - Can be leveraged on successful BD solution implementation
4. Early wins and effort-to-reward ratio
   - Relatively small use case
   - Implemented in short time, smaller budget
   - Optimizes a business critical function
   - Early win

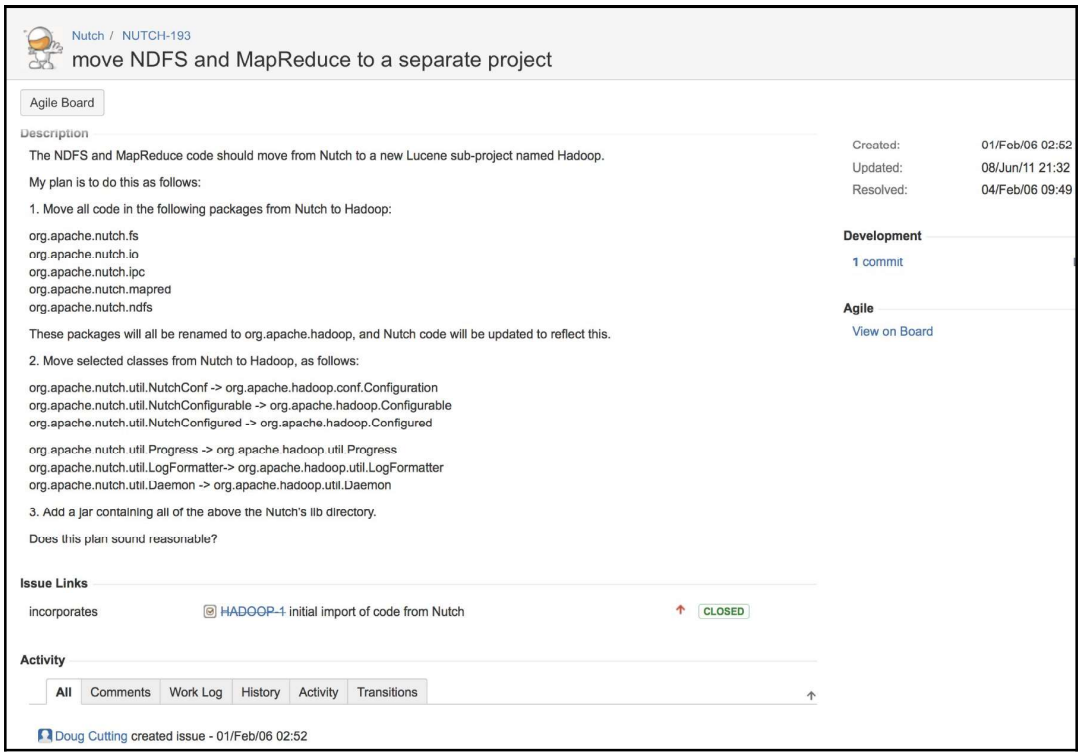
$$\text{E-R Ratio} = \frac{\text{Time} + \text{Cost} + \text{Number of Resources} + \text{Criticality of Use Case}}{\textit{Business Value}}$$

Big Data Mining – Building Corporate Big Data strategy

5. Leveraging the early wins
   - Ground to develop bigger strategy
   - Establishing value of big data to an audience
   - Single department to broader organizational impact

Big Data Mining - Stakeholders

- Business Sponsor
  - Funding department for the project
  - Benefits from the solution
- Implementation group
  - Analysts, Data Scientist, Field workers
- IT procurement
  - Licensing cost
  - Software, Cloud Renting
- Legal
  - Licensing
  - Open source or in-house



Doug Cutting's post at `https://issues.apache.org/jira/browse/NUTCH-193` announced his intent to separate **Nutch Distributed FS** (**NDFS**) and MapReduce to a new subproject called Hadoop.

The fundamental premise of Hadoop

The fundamental premise of Hadoop is that instead of attempting to perform a task on a single large machine, the task can be subdivided into smaller segments that can then be delegated to multiple smaller machines. These so-called smaller machines would then perform the task on their own portion of the data. Once the smaller machines have completed their tasks to produce the results on the tasks they were allocated, the individual units of results would then be aggregated to produce the final result.

Although, in theory, this may appear relatively simple, there are various technical considerations to bear in mind. For example:

- Is the network fast enough to collect the results from each individual server?
- Can each individual server read data fast enough from the disk?
- If one or more of the servers fail, do we have to start all over?
- If there are multiple large tasks, how should they be prioritized?

There are many more such considerations that must be considered when working with a distributed architecture of this nature.

The core modules of Hadoop

The core modules of Hadoop consist of:

- **Hadoop Common**: Libraries and other common helper utilities required by Hadoop
- **HDFS**: A distributed, highly-available, fault-tolerant filesystem that stores data
- **Hadoop MapReduce**: A programming paradigm involving distributed computing across commodity servers (or nodes)
- **Hadoop YARN**: A framework for job scheduling and resource management

Of these core components, YARN was introduced in 2012 to address some of the shortcomings of the first release of Hadoop. The first version of Hadoop (or equivalently, the first model of Hadoop) used HDFS and MapReduce as its main components. As Hadoop gained in popularity, the need to use facilities beyond those provided by MapReduce became more and more important. This, along with some other technical considerations, led to the development of YARN.

Let's now look at the salient characteristics of Hadoop as itemized previously.

Hadoop Distributed File System - HDFS

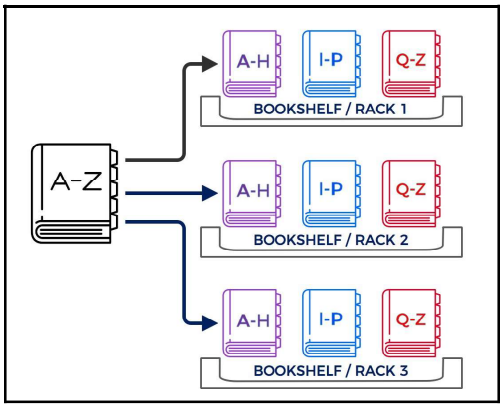
The HDFS forms the underlying basis of all Hadoop installations. Files, or more generally data, is stored in HDFS and accessed by the nodes of Hadoop.

HDFS performs two main functions:

- **Namespaces**: Provides namespaces that hold cluster metadata, that is, the location of data in the Hadoop cluster
- **Data storage**: Acts as storage for data used in the Hadoop cluster

The filesystem is termed as distributed since the data is stored in chunks across multiple servers. An intuitive understanding of HDFS can be gained from a simple example, as follows. Consider a large book that consists of Chapters A - Z. In ordinary filesystems, the entire book would be stored as a single file on the disk. In HDFS, the book would be split into smaller chunks, say a chunk for Chapters A - H, another for I - P, and a third one for Q - Z. These chunks are then stored in separate racks (or bookshelves as with this analogy). Further, the chapters are replicated three times, such that there are three copies of each of the chapters.

Suppose, further, the size of the entire book is 1 GB, and each chapter is approximately 350 MB:



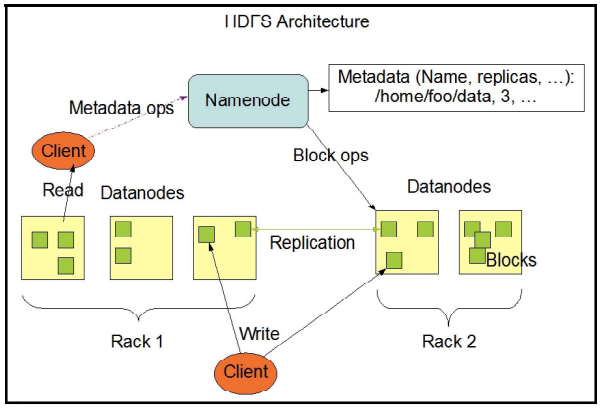
A bookshelf analogy for HDFS

Storing the book in this manner achieves a few important objectives:

- Since the book has been split into three parts by groups of chapters and each part has been replicated three times, it means that our process can read the book in parallel by querying the parts from different servers. This reduces I/O contention and is a very fitting example of the proper use of parallelism.

- If any of the racks are not available, we can retrieve the chapters from any of the other racks as there are multiple copies of each chapter available on different racks.
- If a task I have been given only requires me to access a single chapter, for example, Chapter B, I need to access only the file corresponding to Chapters A-H. Since the size of the file corresponding to Chapters A-H is a third the size of the entire book, the time to access and read the file would be much smaller.
- Other benefits, such as selective access rights to different chapter groups and so on, would also be possible with such a model.

This may be an over-simplified analogy of the actual HDFS functionality, but it conveys the basic principle of the technology - that large files are subdivided into blocks (chunks) and spread across multiple servers in a high-availability redundant configuration. We'll now look at the actual HDFS architecture in a bit more detail:



The HDFS backend of Hadoop consists of:

- **NameNode**: This can be considered the master node. The NameNode contains cluster metadata and is aware of what data is stored in which location - in short, it holds the namespace. It stores the entire namespace in RAM and when a request arrives, provides information on which servers hold the data required for the task. In Hadoop 2, there can be more than one NameNode. A secondary NameNode can be created that acts as a helper node to the primary. As such, it is not a backup NameNode, but one that helps in keeping cluster metadata up to date.

- **DataNode**: The DataNodes are the individual servers that are responsible for storing chunks of the data and performing compute operations when they receive a new request. These are primarily commodity servers that are less powerful in terms of resource and capacity than the NameNode that stores the cluster metadata.

Data storage process in HDFS

The following points should give a good idea of the data storage process:

All data in HDFS is written in blocks, usually of size 128 MB. Thus, a single file of say size 512 MB would be split into four blocks (4 * 128 MB). These blocks are then written to DataNodes. To maintain redundancy and high availability, each block is replicated to create duplicate copies. In general, Hadoop installations have a replication factor of three, indicating that each block of data is replicated three times.

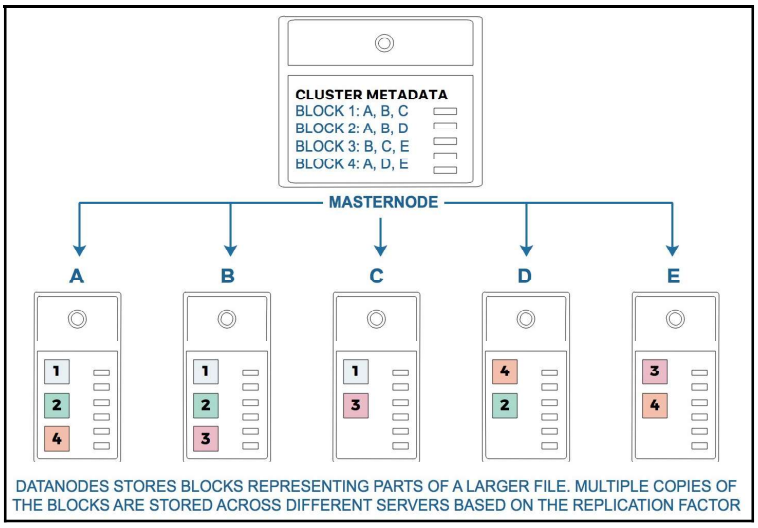
This guarantees redundancy such that in the event one of the servers fails or stops responding, there would always be a second and even a third copy available. To ensure that this process works seamlessly, the DataNode places the replicas in independent servers and can also ensure that the blocks are placed on servers in different racks in a data center. This is due to the fact that even if all the replicas were on independent servers, but all the servers were on the same rack, a rack power failure would mean that no replica would be available.

The general process of writing data into HDFS is as follows:

1. The NameNode receives a request to write a new file to HDFS.
2. Since the data has to be written in blocks or chunks, the HDFS client (the entity that made the request) begins caching data into a local buffer and once the buffer reaches the allocated chunk size (for example, 128 MB), it informs the NameNode that it is ready to write the first block (chunk) of data.
3. The NameNode, based on information available to it about the state of the HDFS cluster, responds with information on the destination DataNode where the block needs to be stored.
4. The HDFS client writes data to the target DataNode and informs the NameNode once the write process for the block has completed.
5. The target DataNode, subsequently, begins copying its copy of the block of data to a second DataNode, which will serve as a replica for the current block.

6. Once the second DataNode completes the write process, it sends the block of data to the third DataNode.
7. This process repeats until all the blocks corresponding to the data (or equivalently, the file) are copied across different nodes.

Note that the number of chunks will depend on the file size. The following image illustrated the distribution of the data across 5 datanodes.



Master Node and Data Nodes

The HDFS architecture in the first release of Hadoop, also known as Hadoop 1, had the following characteristics:

- Single NameNode: Only one NameNode was available, and as a result it also acted as a single point of failure since it stored all the cluster metadata.
- Multiple DataNodes that stored blocks of data, processed client requests, and performed I/O operations (create, read, delete, and so on) on the blocks.
- The HDFS architecture in the second release of Hadoop, also known as Hadoop 2, provided all the benefits of the original HDFS design and also added some new features, most notably, the ability to have multiple NameNodes that can act as primary and secondary NameNodes. Other features included the facility to have multiple namespaces as well as HDFS Federation.

- HDFS Federation deserves special mention. The following excerpt from `http://hadoop.apache.org` explains the subject in a very precise manner:

> *The NameNodes are federated; the NameNodes are independent and do not require coordination with each other. The DataNodes are used as common storage for blocks by all the NameNodes. Each DataNode registers with all the NameNodes in the cluster. DataNodes send periodic heartbeats and block reports.*

The secondary NameNode is not a backup node in the sense that it cannot perform the same tasks as the NameNode in the event that the NameNode is not available. However, it makes the NameNode restart process much more efficient by performing housekeeping operations.

These operations (such as merging HDFS snapshot data with information on data changes) are generally performed by the NameNode when it is restarted and can take a long time depending on the amount of changes since the last restart. The secondary NameNode can, however, perform these housekeeping operations whilst the primary NameNode is still in operation, such that in the event of a restart the primary NameNode can recover much faster. Since the secondary NameNode essentially performs a checkpoint on the HDFS data at periodic intervals, it is also known as the checkpoint node.

Hadoop MapReduce

MapReduce was one of the seminal features of Hadoop that was arguably the most instrumental in bringing it to prominence. MapReduce works on the principle of dividing larger tasks into smaller subtasks. Instead of delegating a single machine to compute a large task, a network of smaller machines can instead be used to complete the smaller subtasks. By distributing the work in this manner, the task can be completed much more efficiently relative to using a single-machine architecture.

This is not dissimilar to how we go about completing work in our day-to-day lives. An example will help to make this clearer.

- **Handling of node failures:** Large Hadoop clusters can contain tens of thousands of nodes. Hence it is likely that there would be server failures on any given day. So that the NameNode is aware of the status of all nodes in the cluster, the DataNodes send a periodic heartbeat to the NameNode. If the NameNode detects that a server has failed, that is, it has stopped receiving heartbeats, it marks the server as failed and replicates all the data that was local to the server onto a new instance.

New features expected in Hadoop 3

At the time of writing this book, Hadoop 3 is in Alpha stage. Details on the new changes that will be available in Hadoop 3 can be found on the internet. For example, `http://hadoop.apache.org/docs/current/` provides the most up-to-date information on new changes to the architecture.

The Hadoop ecosystem

This chapter should be titled as the Apache ecosystem. Hadoop, like all the other projects that will be discussed in this section, is an Apache project. Apache is used loosely as a short form for the open source projects that are supported by the Apache Software Foundation. It originally has its roots in the development of the Apache HTTP server in the early 90s, and today is a collaborative global initiative that comprises entirely of volunteers who participate in releasing open source software to the global technical community.

Hadoop started out as, and still is, one of the projects in the Apache ecosystem. Due to its popularity, many other projects that are also part of Apache have been linked directly or indirectly to Hadoop as they support key functionalities in the Hadoop environment. That said, it is important to bear in mind that these projects can in most cases exist as independent products that can function without a Hadoop environment. Whether it would provide optimal functionality would be a separate topic.

In this section, we'll go over some of the Apache projects that have had a great deal of influence as well as an impact on the growth and usability of Hadoop as a standard IT enterprise solution, as detailed in the following figure:

Product	**Functionality**
Apache Pig	Apache Pig, also known as Pig Latin, is a language specifically designed to represent MapReduce programs through concise statements that define workflows. Coding MapReduce programs in the traditional methods, such as with Java, can be quite complex, and Pig provides an easy abstraction to express a MapReduce workflow and complex **Extract-Transform-Load** (**ETL**) through the use of simple semantics. Pig programs are executed via the Grunt shell.
Apache HBase	Apache HBase is a distributed column-oriented database that sits on top of HDFS. It was modelled on Google's BigTable whereby data is represented in a columnar format. HBase supports low-latency read-write across tables with billions of records and is well suited to tasks that require direct random access to data. More concretely, HBase indexes data in three dimensions - row, column, and timestamp. It also provides a means to represent data with an arbitrary number of columns as column values can be expressed as key-value pairs within the cells of an HBase table.
Apache Hive	Apache Hive provides a SQL-like dialect to query data stored in HDFS. Hive stores data as serialized binary files in a folder-like structure in HDFS. Similar to tables in traditional database management systems, Hive stores data in tabular format in HDFS partitioned based on user-selected attributes. Partitions are thus subfolders of the higher-level directories or tables. There is a third level of abstraction provided by the concept of buckets, which reference files in the partitions of the Hive tables.
Apache Sqoop	Sqoop is used to extract data from traditional databases to HDFS. Large enterprises that have data stored in relational database management systems can thus use Sqoop to transfer data from their data warehouse to a Hadoop implementation.
Apache Flume	Flume is used for the management, aggregation, and analysis of large-scale log data.
Apache Kafka	Kafka is a publish/subscribe-based middleware system that can be used to analyze and subsequently persist (in HDFS) streaming data in real time.

Apache Oozie	Oozie is a workflow management system designed to schedule Hadoop jobs. It implements a key concept known as a **directed acyclic graph** (**DAG**), which will be discussed in our section on Spark.
Apache Spark	Spark is one of the most significant projects in Apache and was designed to address some of the shortcomings of the HDFS-MapReduce model. It started as a relatively small project at UC Berkeley and evolved rapidly to become one of the most prominent alternatives to using Hadoop for analytical tasks. Spark has seen a widespread adoption across the industry and comprises of various other subprojects that provide additional capabilities such as machine learning, streaming analytics, and others.

Hands-on with CDH

In this section, we will utilize the CDH QuickStart VM to work through some of the topics that have been discussed in the current chapter. The exercises do not have to be necessarily performed in a chronological order and are not dependent upon the completion of any of the other exercises.

We will complete the following exercises in this section:

- WordCount using Hadoop MapReduce
- Working with the HDFS
- Downloading and querying data with Apache Hive

WordCount using Hadoop MapReduce

In this exercise, we will be attempting to count the number of occurrences of each word in one of the longest novels ever written. For the exercise, we have selected the book *Artamène ou le Grand Cyrus* written by Georges and/or Madeleine de Scudéry between 1649-1653. The book is considered to be the second longest novel ever written, per the related list on Wikipedia (`https://en.wikipedia.org/wiki/List_of_longest_novels`). The novel consists of 13,905 pages across 10 volumes and has close to two million words.

To begin, we need to launch the Cloudera Distribution of Hadoop Quickstart VM in VirtualBox and double-click on the Cloudera Quickstart VM instance:

- HDFS Federation deserves special mention. The following excerpt from `http://hadoop.apache.org` explains the subject in a very precise manner:

> *The NameNodes are federated; the NameNodes are independent and do not require coordination with each other. The DataNodes are used as common storage for blocks by all the NameNodes. Each DataNode registers with all the NameNodes in the cluster. DataNodes send periodic heartbeats and block reports.*

The secondary NameNode is not a backup node in the sense that it cannot perform the same tasks as the NameNode in the event that the NameNode is not available. However, it makes the NameNode restart process much more efficient by performing housekeeping operations.

These operations (such as merging HDFS snapshot data with information on data changes) are generally performed by the NameNode when it is restarted and can take a long time depending on the amount of changes since the last restart. The secondary NameNode can, however, perform these housekeeping operations whilst the primary NameNode is still in operation, such that in the event of a restart the primary NameNode can recover much faster. Since the secondary NameNode essentially performs a checkpoint on the HDFS data at periodic intervals, it is also known as the checkpoint node.

Hadoop MapReduce

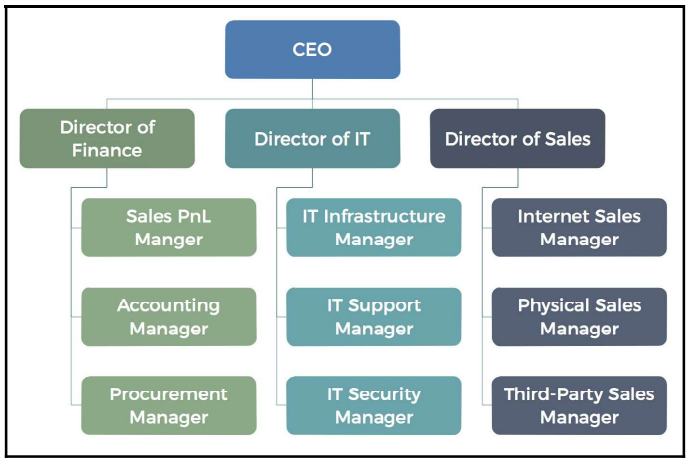
MapReduce was one of the seminal features of Hadoop that was arguably the most instrumental in bringing it to prominence. MapReduce works on the principle of dividing larger tasks into smaller subtasks. Instead of delegating a single machine to compute a large task, a network of smaller machines can instead be used to complete the smaller subtasks. By distributing the work in this manner, the task can be completed much more efficiently relative to using a single-machine architecture.

This is not dissimilar to how we go about completing work in our day-to-day lives. An example will help to make this clearer.

An intuitive introduction to MapReduce

Let's take the example of a hypothetical organization consisting of a CEO, directors, and managers. The CEO wants to know how many new hires have joined the company. The CEO sends a request to his or her directors to report back the number of hires in their departments. The directors in turn send a request to managers in their individual departments to provide the number of new hires. The managers provide the number to the directors, who in turn send the final value back to the CEO.

This can be considered to be a real-world example of MapReduce. In this analogy, the task was finding the number of new hires. Instead of collecting all the data on his or her own, the CEO delegated it to the directors and managers who provided their own individual departmental numbers as illustrated in the following image:



The Concept of MapReduce

In this rather simplistic scenario, the process of splitting a large task (find new hires in the entire company), into smaller tasks (new hires in each team), and then a final re-aggregation of the individual numbers, is analogous to how MapReduce works.

A technical understanding of MapReduce

MapReduce, as the name implies, has a map phase and a reduce phase. A map phase is generally a function that is applied on each element of its input, thus modifying its original value.

MapReduce generates key-value pairs as output.

Key-value: A key-value pair establishes a relationship. For example, if John is 20 years old, a simple key-value pair could be (John, 20). In MapReduce, the map operation produces such key-value pairs that have an entity and the value assigned to the entity.

In practice, map functions can be complex and involve advanced functionalities.

The reduce phase takes the key-value input from the map function and performs a summarization operation. For example, consider the output of a map operation that contains the ages of students in different grades in a school:

Student name	Class	Age
John	Grade 1	7
Mary	Grade 2	8
Jill	Grade 1	6
Tom	Grade 3	10
Mark	Grade 3	9

We can create a simple key-value pair, taking for example the value of Class and Age (it can be anything, but I'm just taking these to provide the example). In this case, our key-value pairs would be (Grade 1, 7), (Grade 2, 8), (Grade 1, 6), (Grade 3, 10), and (Grade 3, 9).

An operation that calculates the average of the ages of students in each grade could then be defined as a reduce operation.

More concretely, we can sort the output and then send the tuples corresponding to each grade to a different server.

For example, Server A would receive the tuples (Grade 1, 7) and (Grade 1, 6), Server B would receive the tuple (Grade 2, 8), Server C would receive the tuples (Grade 3, 10) and (Grade 3, 9). Each of the servers, A, B, and C, would then find the average of the tuples and report back (Grade 1, 6.5), (Grade 2, 8), and (Grade 3, 9.5).

Observe that there was an intermediary step in this process that involved sending the output to a particular server and sorting the output to determine which server it should be sent to. And indeed, MapReduce requires a shuffle and sort phase, whereby the key-value pairs are sorted so that each reducer receives a fixed set of unique keys.

In this example, if say, instead of three servers there were only two, Server A could be assigned to computing averages for keys corresponding to Grades 1 and 2, and Server B could be assigned to computing an average for Grade 3.

In Hadoop, the following process takes place during MapReduce:

1. The client sends a request for a task.
2. NameNode allocates DataNodes (individual servers) that will perform the map operation and ones that will perform the reduce operation. Note that the selection of the DataNode server is dependent upon whether the data that is required for the operation is *local to the server*. The servers where the data resides can only perform the map operation.
3. DataNodes perform the map phase and produce key-value (k,v) pairs.

As the mapper produces the (k,v) pairs, they are sent to these reduce nodes based on the *keys* the node is assigned to compute. The allocation of keys to servers is dependent upon a partitioner function, which could be as simple as a hash value of the key (this is default in Hadoop).

Once the reduce node receives its set of data corresponding to the keys it is responsible to compute on, it applies the reduce function and generates the final output.

Hadoop maximizes the benefits of data locality. Map operations are performed by servers that hold the data locally, that is, on disk. More precisely, the map phase will be executed only by those servers that hold the blocks corresponding to the file. By delegating multiple individual nodes to perform computations independently, the Hadoop architecture can perform very large-scale data processing effectively.

Block size and number of mappers and reducers

An important consideration in the MapReduce process is an understanding of HDFS block size, that is, the size of the chunks into which the files have been split. A MapReduce task that needs to access a certain file will need to perform the map operation on each block representing the file. For example, given a 512 MB file and a 128 MB block size, four blocks would be needed to store the entire file. Hence, a MapReduce operation will at a minimum require four map tasks whereby each map operation would be applied to each subset of the data (that is, each of the four blocks).

If the file was very large, however, and required say, 10,000 blocks to store, this means we would have required 10,000 map operations. But, if we had only 10 servers, then we'd have to send 1,000 map operations to each server. This might be sub-optimal as it can lead to a high penalty due to disk I/O operations and resource allocation settings on a per-map basis.

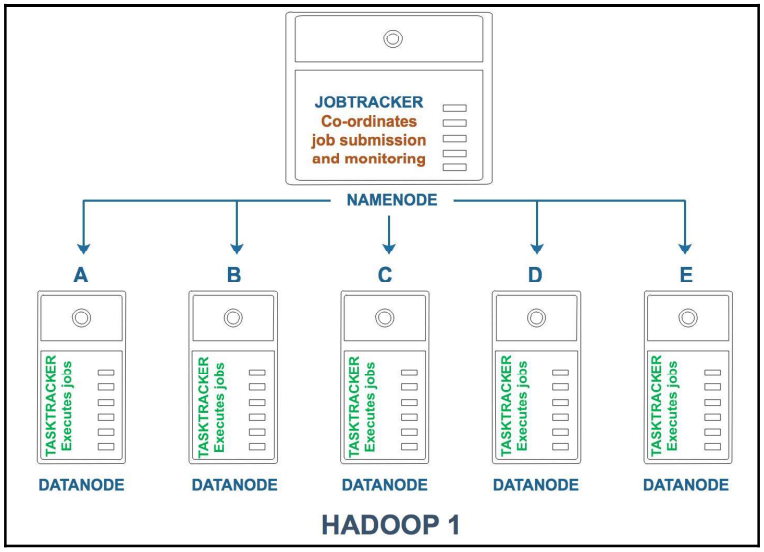
The number of reducers required is summarized very elegantly on Hadoop Wiki (`https://wiki.apache.org/hadoop/HowManyMapsAndReduces`).

> *The ideal reducers should be the optimal value that gets them closest to:*
> ** A multiple of the block size * A task time between 5 and 15 minutes * Creates the fewest files possible*
> *Anything other than that means there is a good chance your reducers are less than great. There is a tremendous tendency for users to use a REALLY high value ("More parallelism means faster!") or a REALLY low value ("I don't want to blow my namespace quota!"). Both are equally dangerous, resulting in one or more of:*
> ** Terrible performance on the next phase of the workflow * Terrible performance due to the shuffle * Terrible overall performance because you've overloaded the namenode with objects that are ultimately useless * Destroying disk IO for no really sane reason * Lots of network transfers due to dealing with crazy amounts of CFIF/MFIF work*

Hadoop YARN

YARN was a module introduced in Hadoop 2. In Hadoop 1, the process of managing jobs and monitoring them was performed by processes known as JobTracker and TaskTracker(s). NameNodes that ran the JobTracker daemon (process) would submit jobs to the DataNodes which ran TaskTracker daemons (processes).

The JobTracker was responsible for the co-ordination of all MapReduce jobs and served as a central administrator for managing processes, handling server failure, re-allocating to new DataNodes, and so on. The TaskTracker monitored the execution of jobs local to its own instance in the DataNode and provided feedback on the status to the JobTracker as shown in the following:



JobTracker and TaskTrackers

This design worked well for a long time, but as Hadoop evolved, the demands for more sophisticated and dynamic functionalities rose proportionally. In Hadoop 1, the NameNode, and consequently the JobTracker process, managed both job scheduling and resource monitoring. In the event the NameNode failed, all activities in the cluster would cease immediately. Lastly, all jobs had to be represented in MapReduce terms - that is, all code would have to be written in the MapReduce framework in order to be executed.

Hadoop 2 alleviated all these concerns:

- The process of job management, scheduling, and resource monitoring was decoupled and delegated to a new framework/module called YARN
- A secondary NameNode could be defined which would act as a helper for the primary NameNode
- Further, Hadoop 2.0 would accommodate frameworks beyond MapReduce
- Instead of fixed map and reduce slots, Hadoop 2 would leverage containers

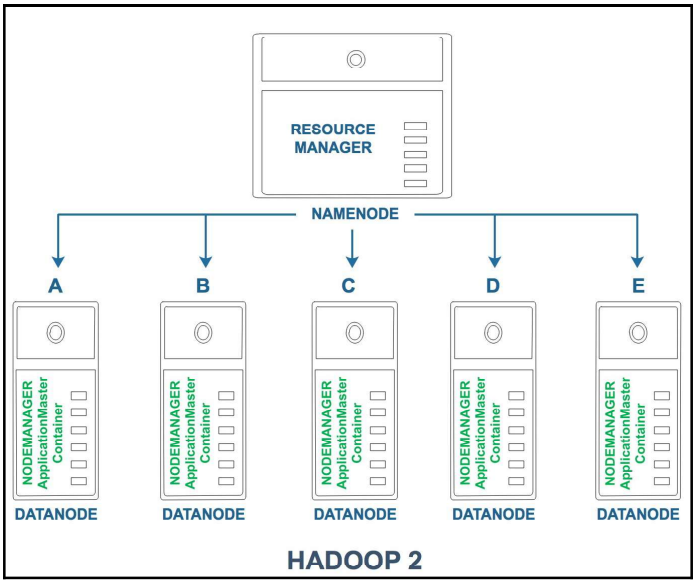
In MapReduce, all data had to be read from disk, and this was fine for operations on large datasets but it was not optimal for operations on smaller datasets. In fact, any tasks that required very fast processing (low latency), were interactive in nature, or had multiple iterations (thus requiring multiple reads from the disk for the same data), would be extremely slow.

By removing these dependencies, Hadoop 2 allowed developers to implement new programming frameworks that would support jobs with diverse performance requirements, such as low latency and interactive real-time querying, iterative processing required for machine learning, different topologies such as the processing of streaming data, optimizations such as in-memory data caching/processing, and so on.

A few new terms became prominent:

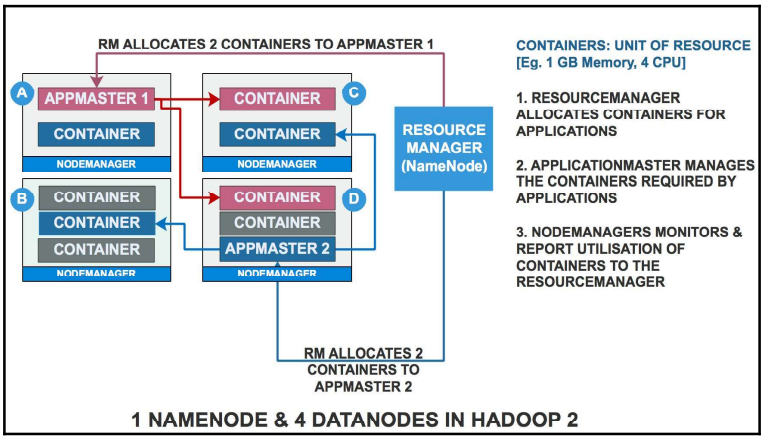
- **ApplicationMaster**: Responsible for managing the resources needed by applications. For example, if a certain job required more memory, the ApplicationMaster would be responsible for securing the required resource. An application in this context refers to application execution frameworks such as MapReduce, Spark, and so on.
- **Containers**: The unit of resource allocation (for example, 1 GB of memory and four CPUs). An application may require several such containers to execute. The ResourceManager allocates containers for executing tasks. Once the allocation is complete, the ApplicationMaster requests DataNodes to start the allocated containers and takes over the management of the containers.
- **ResourceManager**: A component of YARN that had the primary role of allocating resources to applications and functioned as a replacement for the JobTracker. The ResourceManager process ran on the NameNode just as the JobTracker did.
- **NodeManagers**: A replacement for TaskTracker, NodeManagers were responsible for reporting the status of jobs to the ResourceManager (RM) and monitoring the resource utilization of containers.

The following image shows a high level view of the ResourceManager and NodeManagers in Hadoop 2.0:



Hadoop 2.0

The prominent concepts inherent in Hadoop 2 have been illustrated in the next image:



Hadoop 2.0 Concepts

Job scheduling in YARN

It is not uncommon for large Hadoop clusters to have multiple jobs running concurrently. The allocation of resources when there are multiple jobs submitted from multiple departments becomes an important and indeed interesting topic. Which request should receive priority if say, two departments, A and B, submit a job at the same time but each request is for the maximum available resources? In general, Hadoop uses a **First-In-First-Out** (**FIFO**) policy. That is, whoever submits the job first gets to use the resources first. But what if A submitted the job first but completing A's job will take five hours whereas B's job will complete in five minutes?

To deal with these nuances and variables in job scheduling, numerous scheduling methods have been implemented. Three of the more commonly used ones are:

- **FIFO**: As described above, FIFO scheduling uses a queue to priorities jobs. Jobs are executed in the order in which they are submitted.
- **CapacityScheduler**: CapacityScheduler assigns a value on the number of jobs that can be submitted on a per-department basis, where a department can indicate a logical group of users. This is to ensure that each department or group can have access to the Hadoop cluster and be able to utilize a minimum number of resources. The scheduler also allows departments to scale up beyond their assigned capacity up to a maximum value set on a per-department basis if there are unused resources on the server. The model of CapacityScheduler thus provides a guarantee that each department can access the cluster on a deterministic basis.
- **Fair Schedulers**: These schedulers attempt to evenly balance the utilization of resources across different apps. While an even balance might not be feasible at a certain given point in time, balancing allocation over time such that the averages are more or less similar can be achieved using Fair Schedulers.

These, and other schedulers, provide finely grained access controls (such as on a per-user or per-group basis) and primarily utilize queues in order to prioritize and allocate resources.

Other topics in Hadoop

A few other aspects of Hadoop deserve special mention. As we have discussed the most important topics at length, this section provides an overview of some of the other subjects of interest.

CHAPTER 3

The Hadoop Distributed Filesystem

When a dataset outgrows the storage capacity of a single physical machine, it becomes necessary to partition it across a number of separate machines. Filesystems that manage the storage across a network of machines are called *distributed filesystems*. Since they are network based, all the complications of network programming kick in, thus making distributed filesystems more complex than regular disk filesystems. For example, one of the biggest challenges is making the filesystem tolerate node failure without suffering data loss.

Hadoop comes with a distributed filesystem called HDFS, which stands for *Hadoop Distributed Filesystem*. (You may sometimes see references to "DFS"—informally or in older documentation or configurations—which is the same thing.) HDFS is Hadoop's flagship filesystem and is the focus of this chapter, but Hadoop actually has a general-purpose filesystem abstraction, so we'll see along the way how Hadoop integrates with other storage systems (such as the local filesystem and Amazon S3).

The Design of HDFS

HDFS is a filesystem designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware.[1] Let's examine this statement in more detail:

1. The architecture of HDFS is described in Robert Chansler et al.'s, "The Hadoop Distributed File System," which appeared in *The Architecture of Open Source Applications: Elegance, Evolution, and a Few Fearless Hacks* by Amy Brown and Greg Wilson (eds.).

Very large files
: "Very large" in this context means files that are hundreds of megabytes, gigabytes, or terabytes in size. There are Hadoop clusters running today that store petabytes of data.[2]

Streaming data access
: HDFS is built around the idea that the most efficient data processing pattern is a write-once, read-many-times pattern. A dataset is typically generated or copied from source, and then various analyses are performed on that dataset over time. Each analysis will involve a large proportion, if not all, of the dataset, so the time to read the whole dataset is more important than the latency in reading the first record.

Commodity hardware
: Hadoop doesn't require expensive, highly reliable hardware. It's designed to run on clusters of commodity hardware (commonly available hardware that can be obtained from multiple vendors)[3] for which the chance of node failure across the cluster is high, at least for large clusters. HDFS is designed to carry on working without a noticeable interruption to the user in the face of such failure.

It is also worth examining the applications for which using HDFS does not work so well. Although this may change in the future, these are areas where HDFS is not a good fit today:

Low-latency data access
: Applications that require low-latency access to data, in the tens of milliseconds range, will not work well with HDFS. Remember, HDFS is optimized for delivering a high throughput of data, and this may be at the expense of latency. HBase (see Chapter 20) is currently a better choice for low-latency access.

Lots of small files
: Because the namenode holds filesystem metadata in memory, the limit to the number of files in a filesystem is governed by the amount of memory on the namenode. As a rule of thumb, each file, directory, and block takes about 150 bytes. So, for example, if you had one million files, each taking one block, you would need at least 300 MB of memory. Although storing millions of files is feasible, billions is beyond the capability of current hardware.[4]

2. See Konstantin V. Shvachko and Arun C. Murthy, "Scaling Hadoop to 4000 nodes at Yahoo!", September 30, 2008.

3. See Chapter 10 for a typical machine specification.

4. For an exposition of the scalability limits of HDFS, see Konstantin V. Shvachko, "HDFS Scalability: The Limits to Growth", April 2010.

Multiple writers, arbitrary file modifications

Files in HDFS may be written to by a single writer. Writes are always made at the end of the file, in append-only fashion. There is no support for multiple writers or for modifications at arbitrary offsets in the file. (These might be supported in the future, but they are likely to be relatively inefficient.)

HDFS Concepts

Blocks

A disk has a block size, which is the minimum amount of data that it can read or write. Filesystems for a single disk build on this by dealing with data in blocks, which are an integral multiple of the disk block size. Filesystem blocks are typically a few kilobytes in size, whereas disk blocks are normally 512 bytes. This is generally transparent to the filesystem user who is simply reading or writing a file of whatever length. However, there are tools to perform filesystem maintenance, such as *df* and *fsck*, that operate on the filesystem block level.

HDFS, too, has the concept of a block, but it is a much larger unit—128 MB by default. Like in a filesystem for a single disk, files in HDFS are broken into block-sized chunks, which are stored as independent units. Unlike a filesystem for a single disk, a file in HDFS that is smaller than a single block does not occupy a full block's worth of underlying storage. (For example, a 1 MB file stored with a block size of 128 MB uses 1 MB of disk space, not 128 MB.) When unqualified, the term "block" in this book refers to a block in HDFS.

Why Is a Block in HDFS So Large?

HDFS blocks are large compared to disk blocks, and the reason is to minimize the cost of seeks. If the block is large enough, the time it takes to transfer the data from the disk can be significantly longer than the time to seek to the start of the block. Thus, transferring a large file made of multiple blocks operates at the disk transfer rate.

A quick calculation shows that if the seek time is around 10 ms and the transfer rate is 100 MB/s, to make the seek time 1% of the transfer time, we need to make the block size around 100 MB. The default is actually 128 MB, although many HDFS installations use larger block sizes. This figure will continue to be revised upward as transfer speeds grow with new generations of disk drives.

This argument shouldn't be taken too far, however. Map tasks in MapReduce normally operate on one block at a time, so if you have too few tasks (fewer than nodes in the cluster), your jobs will run slower than they could otherwise.

Having a block abstraction for a distributed filesystem brings several benefits. The first benefit is the most obvious: a file can be larger than any single disk in the network. There's nothing that requires the blocks from a file to be stored on the same disk, so they can take advantage of any of the disks in the cluster. In fact, it would be possible, if unusual, to store a single file on an HDFS cluster whose blocks filled all the disks in the cluster.

Second, making the unit of abstraction a block rather than a file simplifies the storage subsystem. Simplicity is something to strive for in all systems, but it is especially important for a distributed system in which the failure modes are so varied. The storage subsystem deals with blocks, simplifying storage management (because blocks are a fixed size, it is easy to calculate how many can be stored on a given disk) and eliminating metadata concerns (because blocks are just chunks of data to be stored, file metadata such as permissions information does not need to be stored with the blocks, so another system can handle metadata separately).

Furthermore, blocks fit well with replication for providing fault tolerance and availability. To insure against corrupted blocks and disk and machine failure, each block is replicated to a small number of physically separate machines (typically three). If a block becomes unavailable, a copy can be read from another location in a way that is transparent to the client. A block that is no longer available due to corruption or machine failure can be replicated from its alternative locations to other live machines to bring the replication factor back to the normal level. (See "Data Integrity" on page 97 for more on guarding against corrupt data.) Similarly, some applications may choose to set a high replication factor for the blocks in a popular file to spread the read load on the cluster.

Like its disk filesystem cousin, HDFS's `fsck` command understands blocks. For example, running:

```
% hdfs fsck / -files -blocks
```

will list the blocks that make up each file in the filesystem. (See also "Filesystem check (fsck)" on page 326.)

Namenodes and Datanodes

An HDFS cluster has two types of nodes operating in a master–worker pattern: a *namenode* (the master) and a number of *datanodes* (workers). The namenode manages the filesystem namespace. It maintains the filesystem tree and the metadata for all the files and directories in the tree. This information is stored persistently on the local disk in the form of two files: the namespace image and the edit log. The namenode also knows the datanodes on which all the blocks for a given file are located; however, it does not store block locations persistently, because this information is reconstructed from datanodes when the system starts.

A *client* accesses the filesystem on behalf of the user by communicating with the namenode and datanodes. The client presents a filesystem interface similar to a Portable Operating System Interface (POSIX), so the user code does not need to know about the namenode and datanodes to function.

Datanodes are the workhorses of the filesystem. They store and retrieve blocks when they are told to (by clients or the namenode), and they report back to the namenode periodically with lists of blocks that they are storing.

Without the namenode, the filesystem cannot be used. In fact, if the machine running the namenode were obliterated, all the files on the filesystem would be lost since there would be no way of knowing how to reconstruct the files from the blocks on the datanodes. For this reason, it is important to make the namenode resilient to failure, and Hadoop provides two mechanisms for this.

The first way is to back up the files that make up the persistent state of the filesystem metadata. Hadoop can be configured so that the namenode writes its persistent state to multiple filesystems. These writes are synchronous and atomic. The usual configuration choice is to write to local disk as well as a remote NFS mount.

It is also possible to run a *secondary namenode*, which despite its name does not act as a namenode. Its main role is to periodically merge the namespace image with the edit log to prevent the edit log from becoming too large. The secondary namenode usually runs on a separate physical machine because it requires plenty of CPU and as much memory as the namenode to perform the merge. It keeps a copy of the merged namespace image, which can be used in the event of the namenode failing. However, the state of the secondary namenode lags that of the primary, so in the event of total failure of the primary, data loss is almost certain. The usual course of action in this case is to copy the namenode's metadata files that are on NFS to the secondary and run it as the new primary. (Note that it is possible to run a hot standby namenode instead of a secondary, as discussed in "HDFS High Availability" on page 48.)

See "The filesystem image and edit log" on page 318 for more details.

Block Caching

Normally a datanode reads blocks from disk, but for frequently accessed files the blocks may be explicitly cached in the datanode's memory, in an off-heap *block cache*. By default, a block is cached in only one datanode's memory, although the number is configurable on a per-file basis. Job schedulers (for MapReduce, Spark, and other frameworks) can take advantage of cached blocks by running tasks on the datanode where a block is cached, for increased read performance. A small lookup table used in a join is a good candidate for caching, for example.

Users or applications instruct the namenode which files to cache (and for how long) by adding a *cache directive* to a *cache pool*. Cache pools are an administrative grouping for managing cache permissions and resource usage.

HDFS Federation

The namenode keeps a reference to every file and block in the filesystem in memory, which means that on very large clusters with many files, memory becomes the limiting factor for scaling (see "How Much Memory Does a Namenode Need?" on page 294). HDFS federation, introduced in the 2.x release series, allows a cluster to scale by adding namenodes, each of which manages a portion of the filesystem namespace. For example, one namenode might manage all the files rooted under */user*, say, and a second namenode might handle files under */share*.

Under federation, each namenode manages a *namespace volume*, which is made up of the metadata for the namespace, and a *block pool* containing all the blocks for the files in the namespace. Namespace volumes are independent of each other, which means namenodes do not communicate with one another, and furthermore the failure of one namenode does not affect the availability of the namespaces managed by other namenodes. Block pool storage is *not* partitioned, however, so datanodes register with each namenode in the cluster and store blocks from multiple block pools.

To access a federated HDFS cluster, clients use client-side mount tables to map file paths to namenodes. This is managed in configuration using `ViewFileSystem` and the `viewfs://` URIs.

HDFS High Availability

The combination of replicating namenode metadata on multiple filesystems and using the secondary namenode to create checkpoints protects against data loss, but it does not provide high availability of the filesystem. The namenode is still a *single point of failure* (SPOF). If it did fail, all clients—including MapReduce jobs—would be unable to read, write, or list files, because the namenode is the sole repository of the metadata and the file-to-block mapping. In such an event, the whole Hadoop system would effectively be out of service until a new namenode could be brought online.

To recover from a failed namenode in this situation, an administrator starts a new primary namenode with one of the filesystem metadata replicas and configures datanodes and clients to use this new namenode. The new namenode is not able to serve requests until it has (i) loaded its namespace image into memory, (ii) replayed its edit log, and (iii) received enough block reports from the datanodes to leave safe mode. On large clusters with many files and blocks, the time it takes for a namenode to start from cold can be 30 minutes or more.

The long recovery time is a problem for routine maintenance, too. In fact, because unexpected failure of the namenode is so rare, the case for planned downtime is actually more important in practice.

Hadoop 2 remedied this situation by adding support for HDFS high availability (HA). In this implementation, there are a pair of namenodes in an active-standby configuration. In the event of the failure of the active namenode, the standby takes over its duties to continue servicing client requests without a significant interruption. A few architectural changes are needed to allow this to happen:

- The namenodes must use highly available shared storage to share the edit log. When a standby namenode comes up, it reads up to the end of the shared edit log to synchronize its state with the active namenode, and then continues to read new entries as they are written by the active namenode.
- Datanodes must send block reports to both namenodes because the block mappings are stored in a namenode's memory, and not on disk.
- Clients must be configured to handle namenode failover, using a mechanism that is transparent to users.
- The secondary namenode's role is subsumed by the standby, which takes periodic checkpoints of the active namenode's namespace.

There are two choices for the highly available shared storage: an NFS filer, or a *quorum journal manager* (QJM). The QJM is a dedicated HDFS implementation, designed for the sole purpose of providing a highly available edit log, and is the recommended choice for most HDFS installations. The QJM runs as a group of *journal nodes*, and each edit must be written to a majority of the journal nodes. Typically, there are three journal nodes, so the system can tolerate the loss of one of them. This arrangement is similar to the way ZooKeeper works, although it is important to realize that the QJM implementation does not use ZooKeeper. (Note, however, that HDFS HA *does* use ZooKeeper for electing the active namenode, as explained in the next section.)

If the active namenode fails, the standby can take over very quickly (in a few tens of seconds) because it has the latest state available in memory: both the latest edit log entries and an up-to-date block mapping. The actual observed failover time will be longer in practice (around a minute or so), because the system needs to be conservative in deciding that the active namenode has failed.

In the unlikely event of the standby being down when the active fails, the administrator can still start the standby from cold. This is no worse than the non-HA case, and from an operational point of view it's an improvement, because the process is a standard operational procedure built into Hadoop.

Failover and fencing

The transition from the active namenode to the standby is managed by a new entity in the system called the *failover controller*. There are various failover controllers, but the default implementation uses ZooKeeper to ensure that only one namenode is active. Each namenode runs a lightweight failover controller process whose job it is to monitor its namenode for failures (using a simple heartbeating mechanism) and trigger a failover should a namenode fail.

Failover may also be initiated manually by an administrator, for example, in the case of routine maintenance. This is known as a *graceful failover*, since the failover controller arranges an orderly transition for both namenodes to switch roles.

In the case of an ungraceful failover, however, it is impossible to be sure that the failed namenode has stopped running. For example, a slow network or a network partition can trigger a failover transition, even though the previously active namenode is still running and thinks it is still the active namenode. The HA implementation goes to great lengths to ensure that the previously active namenode is prevented from doing any damage and causing corruption—a method known as *fencing*.

The QJM only allows one namenode to write to the edit log at one time; however, it is still possible for the previously active namenode to serve stale read requests to clients, so setting up an SSH fencing command that will kill the namenode's process is a good idea. Stronger fencing methods are required when using an NFS filer for the shared edit log, since it is not possible to only allow one namenode to write at a time (this is why QJM is recommended). The range of fencing mechanisms includes revoking the namenode's access to the shared storage directory (typically by using a vendor-specific NFS command), and disabling its network port via a remote management command. As a last resort, the previously active namenode can be fenced with a technique rather graphically known as *STONITH*, or "shoot the other node in the head," which uses a specialized power distribution unit to forcibly power down the host machine.

Client failover is handled transparently by the client library. The simplest implementation uses client-side configuration to control failover. The HDFS URI uses a logical hostname that is mapped to a pair of namenode addresses (in the configuration file), and the client library tries each namenode address until the operation succeeds.

The Command-Line Interface

We're going to have a look at HDFS by interacting with it from the command line. There are many other interfaces to HDFS, but the command line is one of the simplest and, to many developers, the most familiar.

We are going to run HDFS on one machine, so first follow the instructions for setting up Hadoop in pseudodistributed mode in Appendix A. Later we'll see how to run HDFS on a cluster of machines to give us scalability and fault tolerance.

Data Flow

Anatomy of a File Read

To get an idea of how data flows between the client interacting with HDFS, the namenode, and the datanodes, consider Figure 3-2, which shows the main sequence of events when reading a file.

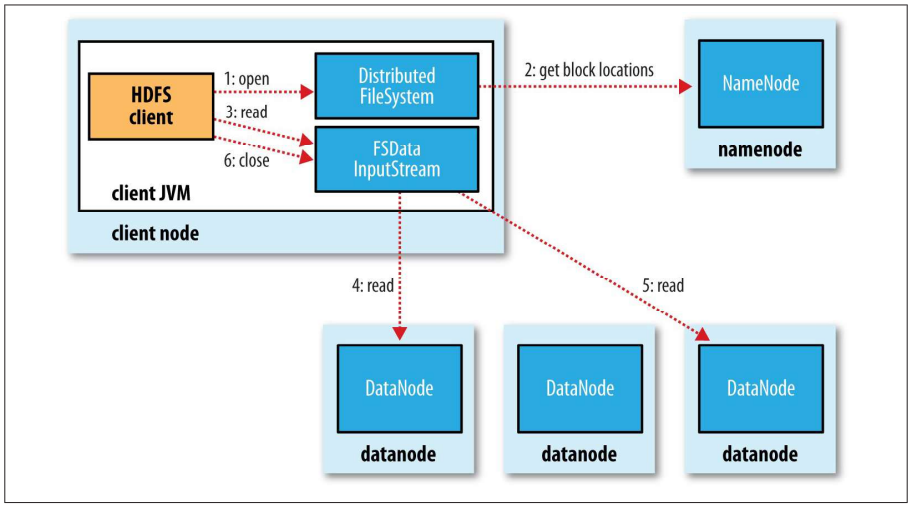


Figure 3-2. A client reading data from HDFS

The client opens the file it wishes to read by calling `open()` on the `FileSystem` object, which for HDFS is an instance of `DistributedFileSystem` (step 1 in Figure 3-2). `DistributedFileSystem` calls the namenode, using remote procedure calls (RPCs), to determine the locations of the first few blocks in the file (step 2). For each block, the namenode returns the addresses of the datanodes that have a copy of that block. Furthermore, the datanodes are sorted according to their proximity to the client (according to the topology of the cluster's network; see "Network Topology and Hadoop" on page 70). If the client is itself a datanode (in the case of a MapReduce task, for instance), the client will read from the local datanode if that datanode hosts a copy of the block (see also Figure 2-2 and "Short-circuit local reads" on page 308).

The `DistributedFileSystem` returns an `FSDataInputStream` (an input stream that supports file seeks) to the client for it to read data from. `FSDataInputStream` in turn wraps a `DFSInputStream`, which manages the datanode and namenode I/O.

The client then calls `read()` on the stream (step 3). `DFSInputStream`, which has stored the datanode addresses for the first few blocks in the file, then connects to the first

(closest) datanode for the first block in the file. Data is streamed from the datanode back to the client, which calls `read()` repeatedly on the stream (step 4). When the end of the block is reached, `DFSInputStream` will close the connection to the datanode, then find the best datanode for the next block (step 5). This happens transparently to the client, which from its point of view is just reading a continuous stream.

Blocks are read in order, with the `DFSInputStream` opening new connections to datanodes as the client reads through the stream. It will also call the namenode to retrieve the datanode locations for the next batch of blocks as needed. When the client has finished reading, it calls `close()` on the `FSDataInputStream` (step 6).

During reading, if the `DFSInputStream` encounters an error while communicating with a datanode, it will try the next closest one for that block. It will also remember datanodes that have failed so that it doesn't needlessly retry them for later blocks. The `DFSInputStream` also verifies checksums for the data transferred to it from the datanode. If a corrupted block is found, the `DFSInputStream` attempts to read a replica of the block from another datanode; it also reports the corrupted block to the namenode.

One important aspect of this design is that the client contacts datanodes directly to retrieve data and is guided by the namenode to the best datanode for each block. This design allows HDFS to scale to a large number of concurrent clients because the data traffic is spread across all the datanodes in the cluster. Meanwhile, the namenode merely has to service block location requests (which it stores in memory, making them very efficient) and does not, for example, serve data, which would quickly become a bottleneck as the number of clients grew.

Network Topology and Hadoop

What does it mean for two nodes in a local network to be "close" to each other? In the context of high-volume data processing, the limiting factor is the rate at which we can transfer data between nodes—bandwidth is a scarce commodity. The idea is to use the bandwidth between two nodes as a measure of distance.

Rather than measuring bandwidth between nodes, which can be difficult to do in practice (it requires a quiet cluster, and the number of pairs of nodes in a cluster grows as the square of the number of nodes), Hadoop takes a simple approach in which the network is represented as a tree and the distance between two nodes is the sum of their distances to their closest common ancestor. Levels in the tree are not predefined, but it is common to have levels that correspond to the data center, the rack, and the node that a process is running on. The idea is that the bandwidth available for each of the following scenarios becomes progressively less:

- Processes on the same node
- Different nodes on the same rack

- Nodes on different racks in the same data center
- Nodes in different data centers[8]

For example, imagine a node *n1* on rack *r1* in data center *d1*. This can be represented as */d1/r1/n1*. Using this notation, here are the distances for the four scenarios:

- *distance(/d1/r1/n1, /d1/r1/n1)* = 0 (processes on the same node)
- *distance(/d1/r1/n1, /d1/r1/n2)* = 2 (different nodes on the same rack)
- *distance(/d1/r1/n1, /d1/r2/n3)* = 4 (nodes on different racks in the same data center)
- *distance(/d1/r1/n1, /d2/r3/n4)* = 6 (nodes in different data centers)

This is illustrated schematically in Figure 3-3. (Mathematically inclined readers will notice that this is an example of a distance metric.)

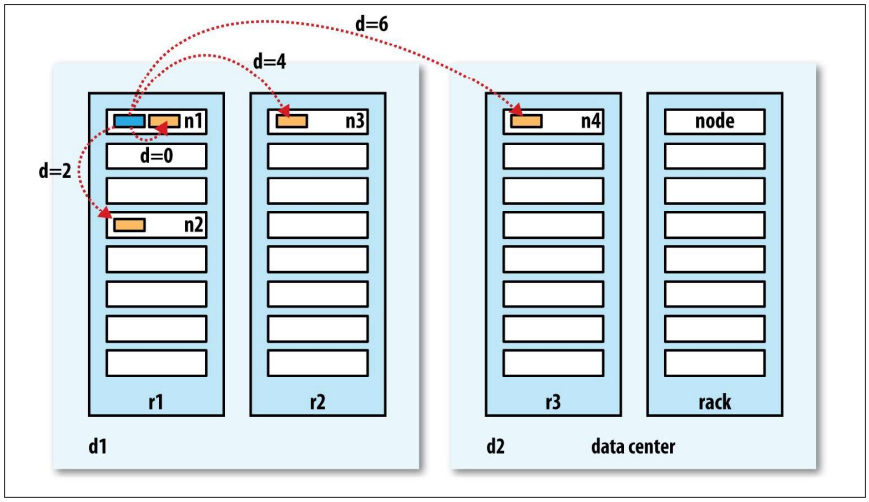


Figure 3-3. Network distance in Hadoop

Finally, it is important to realize that Hadoop cannot magically discover your network topology for you; it needs some help (we'll cover how to configure topology in "Network Topology" on page 286). By default, though, it assumes that the network is flat—a single-level hierarchy—or in other words, that all nodes are on a single rack in a single data center. For small clusters, this may actually be the case, and no further configuration is required.

8. At the time of this writing, Hadoop is not suited for running across data centers.

Anatomy of a File Write

Next we'll look at how files are written to HDFS. Although quite detailed, it is instructive to understand the data flow because it clarifies HDFS's coherency model.

We're going to consider the case of creating a new file, writing data to it, then closing the file. This is illustrated in Figure 3-4.

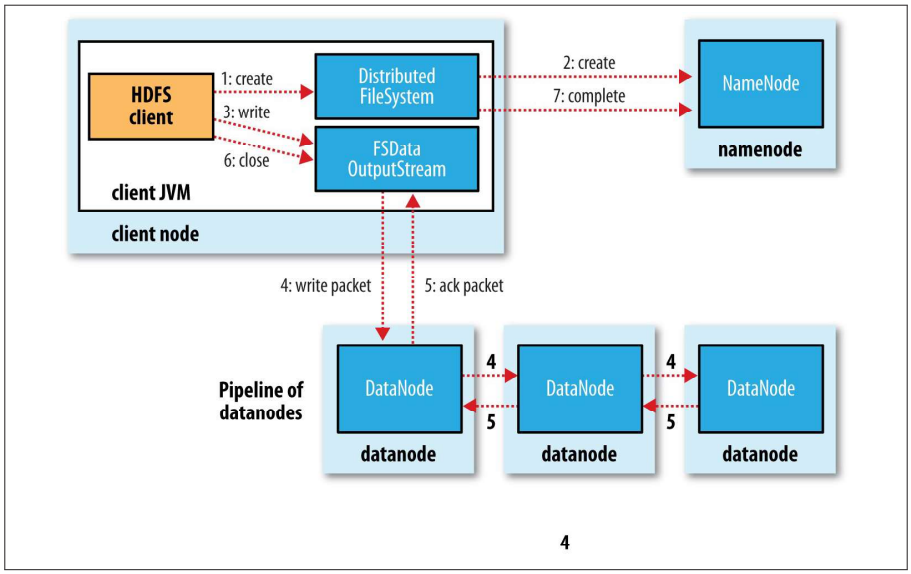


Figure 3-4. A client writing data to HDFS

The client creates the file by calling `create()` on `DistributedFileSystem` (step 1 in Figure 3-4). `DistributedFileSystem` makes an RPC call to the namenode to create a new file in the filesystem's namespace, with no blocks associated with it (step 2). The namenode performs various checks to make sure the file doesn't already exist and that the client has the right permissions to create the file. If these checks pass, the namenode makes a record of the new file; otherwise, file creation fails and the client is thrown an `IOException`. The `DistributedFileSystem` returns an `FSDataOutputStream` for the client to start writing data to. Just as in the read case, `FSDataOutputStream` wraps a `DFSOutputStream`, which handles communication with the datanodes and namenode.

As the client writes data (step 3), the `DFSOutputStream` splits it into packets, which it writes to an internal queue called the *data queue*. The data queue is consumed by the `DataStreamer`, which is responsible for asking the namenode to allocate new blocks by picking a list of suitable datanodes to store the replicas. The list of datanodes forms a pipeline, and here we'll assume the replication level is three, so there are three nodes in

the pipeline. The `DataStreamer` streams the packets to the first datanode in the pipeline, which stores each packet and forwards it to the second datanode in the pipeline. Similarly, the second datanode stores the packet and forwards it to the third (and last) datanode in the pipeline (step 4).

The `DFSOutputStream` also maintains an internal queue of packets that are waiting to be acknowledged by datanodes, called the *ack queue*. A packet is removed from the ack queue only when it has been acknowledged by all the datanodes in the pipeline (step 5).

If any datanode fails while data is being written to it, then the following actions are taken, which are transparent to the client writing the data. First, the pipeline is closed, and any packets in the ack queue are added to the front of the data queue so that datanodes that are downstream from the failed node will not miss any packets. The current block on the good datanodes is given a new identity, which is communicated to the namenode, so that the partial block on the failed datanode will be deleted if the failed datanode recovers later on. The failed datanode is removed from the pipeline, and a new pipeline is constructed from the two good datanodes. The remainder of the block's data is written to the good datanodes in the pipeline. The namenode notices that the block is under-replicated, and it arranges for a further replica to be created on another node. Subsequent blocks are then treated as normal.

It's possible, but unlikely, for multiple datanodes to fail while a block is being written. As long as `dfs.namenode.replication.min` replicas (which defaults to 1) are written, the write will succeed, and the block will be asynchronously replicated across the cluster until its target replication factor is reached (`dfs.replication`, which defaults to 3).

When the client has finished writing data, it calls `close()` on the stream (step 6). This action flushes all the remaining packets to the datanode pipeline and waits for acknowledgments before contacting the namenode to signal that the file is complete (step 7). The namenode already knows which blocks the file is made up of (because `DataStreamer` asks for block allocations), so it only has to wait for blocks to be minimally replicated before returning successfully.

Replica Placement

How does the namenode choose which datanodes to store replicas on? There's a trade-off between reliability and write bandwidth and read bandwidth here. For example, placing all replicas on a single node incurs the lowest write bandwidth penalty (since the replication pipeline runs on a single node), but this offers no real redundancy (if the node fails, the data for that block is lost). Also, the read bandwidth is high for off-rack reads. At the other extreme, placing replicas in different data centers may maximize redundancy, but at the cost of bandwidth. Even in the same data center (which is what all Hadoop clusters to date have run in), there are a variety of possible placement strategies.

Hadoop's default strategy is to place the first replica on the same node as the client (for clients running outside the cluster, a node is chosen at random, although the system tries not to pick nodes that are too full or too busy). The second replica is placed on a different rack from the first (*off-rack*), chosen at random. The third replica is placed on the same rack as the second, but on a different node chosen at random. Further replicas are placed on random nodes in the cluster, although the system tries to avoid placing too many replicas on the same rack.

Once the replica locations have been chosen, a pipeline is built, taking network topology into account. For a replication factor of 3, the pipeline might look like Figure 3-5.

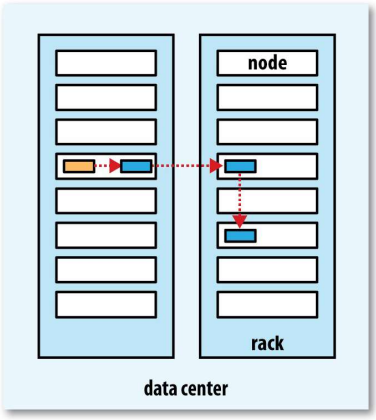


Figure 3-5. A typical replica pipeline

Overall, this strategy gives a good balance among reliability (blocks are stored on two racks), write bandwidth (writes only have to traverse a single network switch), read performance (there's a choice of two racks to read from), and block distribution across the cluster (clients only write a single block on the local rack).

Coherency Model

A coherency model for a filesystem describes the data visibility of reads and writes for a file. HDFS trades off some POSIX requirements for performance, so some operations may behave differently than you expect them to.

After creating a file, it is visible in the filesystem namespace, as expected:

```
Path p = new Path("p");
fs.create(p);
assertThat(fs.exists(p), is(true));
```