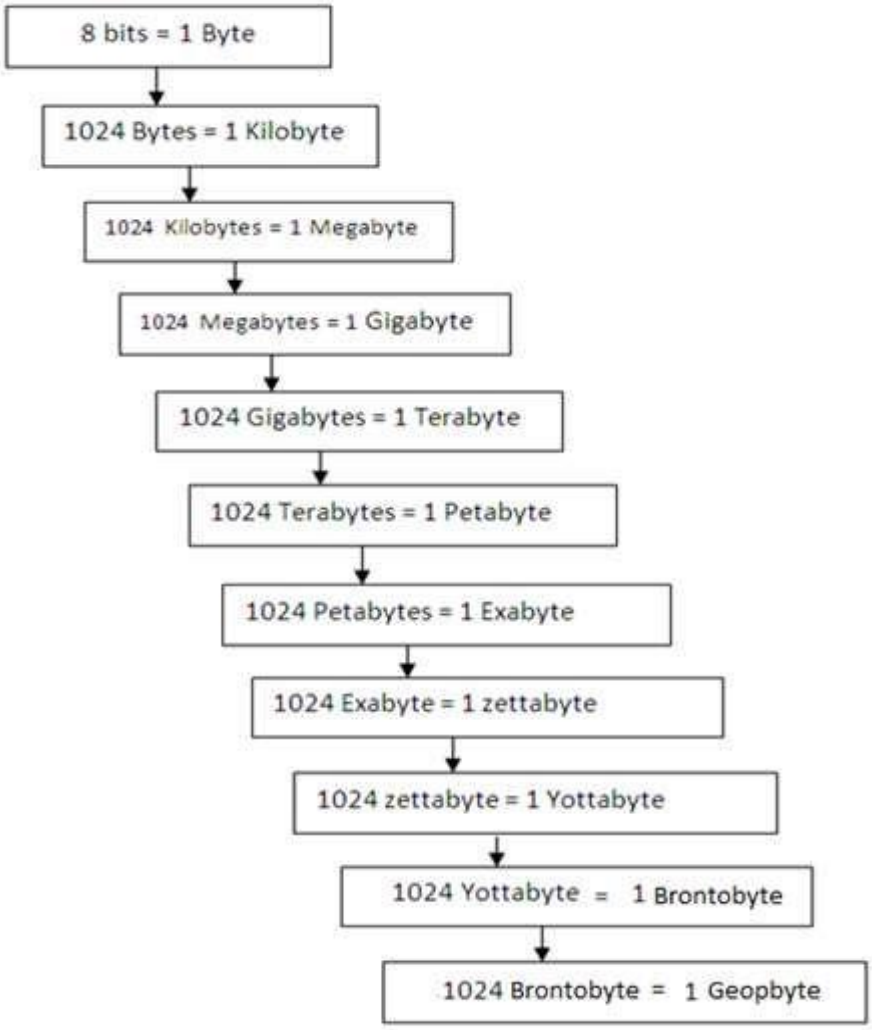
BIG DATA ANALYTICS

1.Introduction to Big Data:

What is Big Data and Big Data Analytics (BDA)?

"***Big Data*** is an evolving term that describes any large amount of structured, semi-structured and unstructured data that has the potential to be mined for information."

"Big Data Analytics (BDA) is the process of examining **large data** sets containing a variety of **data** types -- i.e., **big data** -- to uncover hidden patterns, unknown correlations,



market trends, customer preferences and other useful business information."

Characteristics of Big Data (or) Why is Big Data different from any other data?

There are "Five V"s" that characterize this data: Volume, Velocity, Variety, Veracity and Validity.

1. Volume (Data Quantity):

Most organizations were already struggling with the increasing size of their databases as the Big Data tsunami hit the data stores.

2. Velocity (Data Speed):

There are two aspects to velocity. They are throughput of data and the other representing latency.

a. Throughput which represents the data moving in the pipes.

b. Latency is the other measure of velocity. Analytics used to be a "store and report" environment where reporting typically contained data as of yesterday—popularly represented as "D-1."

c. Variety (Data Types):

The source data includes unstructured text, sound, and video in addition to structured data. A number of applications are gathering data from emails, documents, or blogs.

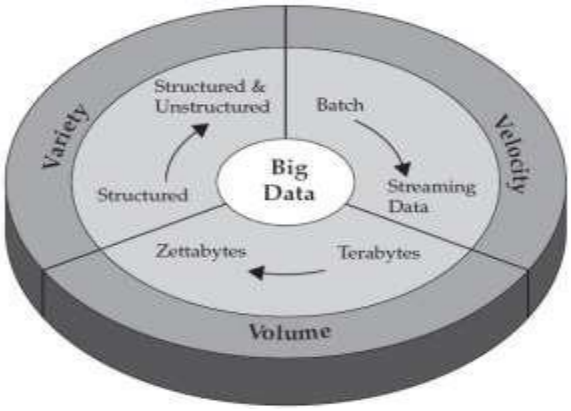
d. Veracity (Data Quality):

Veracity represents both the credibility of the data source as well as the suitability of the data for the target audience.

e. Validity (Data Correctness):

Validity meaning is the data correct and accurate for the future use. Clearly valid data is key to making the right decisions.

As per IBM, the number of characteristics of Big Data is V^3 and described in the following Figure:



Importance of Big Data:

1. Access to social data from search engines and sites like Facebook, twitter are enabling organizations to fine tune their business strategies. Marketing agencies are learning about the response for their campaigns, promotions, and other advertising mediums.
2. Traditional customer feedback systems are getting replaced by new systems designed with „Big Data" technologies. In these new systems, Big Data and natural language processing technologies are being used to read and evaluate consumer responses.
3. Based on information in the social media like preferences and product perception of their consumers, product companies and retail organizations are planning their

production.

4. Determining root causes of failures, issues and defects in near-real time
5. Detecting fraudulent behavior before it affects the organization.

→When to you use Big Data technologies?

1. Big Data solutions are ideal for analyzing not only raw structured data, but semi-structured and unstructured data from a wide variety of sources.
2. Big Data solutions are ideal when all, or most, of the data needs to be analyzed versus a sample of the data; or a sampling of data isn‟t nearly as effective as a larger set of data from which to derive analysis.
3. Big Data solutions are ideal for iterative and exploratory analysis when business measures on data are not predetermined.

Patterns for Big Data Development:

The following six most common usage patterns represent great Big Data opportunities—business problems that weren‟t easy to solve before—and help us gain an understanding of how Big Data can help us (or how it‟s helping our competitors make us less competitive if we are not paying attention).

1. IT for IT Log Analytics

2. The Fraud Detection Pattern

3. The Social Media Pattern

4. The Call Center Mantra: "This Call May Be Recorded for Quality Assurance Purposes"

5. Risk: Patterns for Modeling and Management

6. Big Data and the Energy Sector

1. IT for IT Log Analytics: IT departments need logs at their disposal, and today they just can‟t store enough logs and analyze them in a cost-efficient manner, so logs are typically kept for emergencies and discarded as soon as possible. Another reason why IT departments keep large amounts of data in logs is to look for rare problems. It is often the case that the most common problems are known and easy to deal with, but the problem that happens "once in a while" is typically more difficult to diagnose and prevent from occurring again.

But there are more reasons why log analysis is a Big Data problem apart from its large nature. The nature of these logs is semi-structured and raw, so they aren‟t always suited for traditional database processing. In addition, log formats are constantly changing due to hardware and software upgrades, so they can‟t be tied to strict inflexible analysis paradigms.

Finally, not only do we need to perform analysis on the longevity of the logs to determine trends and patterns and to find failures, but also we need to ensure the analysis is done on all the data.

Log analytics is actually a pattern that IBM established after working with a number of companies, including some large financial services sector (FSS) companies. This use case comes up with quite a few customers since; for that reason, this pattern is called *IT for IT*. If we are new to this usage pattern and wondering just who is interested in IT for IT Big Data solutions, we should know that this is an internal use case within an organization itself. An internal IT for IT implementation is well suited for any organization with a large data center footprint, especially if it is relatively complex. For example, service-oriented architecture (SOA) applications with lots of moving parts, federated data centers, and so on, all suffer from the same issues outlined in this section.

Some of large insurance and retail clients need to know the answers to such questions as, "What are the precursors to failures?", "How are these systems all related?", and more. These are the types of questions that conventional monitoring doesn"t answer; a Big Data platform finally offers the opportunity to get some new and better insights into the problems at hand.

2. The Fraud Detection Pattern: Fraud detection comes up a lot in the financial services vertical, we will find it in any sort of claims- or transaction-based environment (online auctions, insurance claims, underwriting entities, and so on). Pretty much anywhere some sortof financial transaction is involved presents a potential for misuse and the universal threat of fraud. If we influence a Big Data platform, we have the opportunity to do more than we have ever done before to identify it or, better yet, stop it.

Traditionally, in fraud cases, samples and models are used to identify customers that characterize a certain kind of profile. The problem with this is that although it works, we are profiling a segment and not the granularity at an individual transaction or person level. As per customer experiences, it is estimated that only 20 percent (or maybe less) of the available information that could be useful for fraud modeling is actually being used. The traditional approach is shown in the following Figure.

Finally, not only do we need to perform analysis on the longevity of the logs to determine trends and patterns and to find failures, but also we need to ensure the analysis is done on all the data.

Log analytics is actually a pattern that IBM established after working with a number of companies, including some large financial services sector (FSS) companies. This use case

comes up with quite a few customers since; for that reason, this pattern is called *IT for IT*. If we are new to this usage pattern and wondering just who is interested in IT for IT Big Data solutions, we should know that this is an internal use case within an organization itself. An internal IT for IT implementation is well suited for any organization with a large data center footprint, especially if it is relatively complex. For example, service-oriented architecture (SOA) applications with lots of moving parts, federated data centers, and so on, all suffer from the same issues outlined in this section.

Some of large insurance and retail clients need to know the answers to such questions as, "What are the precursors to failures?", "How are these systems all related?", and more. These are the types of questions that conventional monitoring doesn"t answer; a Big Data platform finally offers the opportunity to get some new and better insights into the problems at hand.

3. The Fraud Detection Pattern: Fraud detection comes up a lot in the financial services vertical, we will find it in any sort of claims- or transaction-based environment (online auctions, insurance claims, underwriting entities, and so on). Pretty much anywhere some sortof financial transaction is involved presents a potential for misuse and the universal threat of fraud. If we influence a Big Data platform, we have the opportunity to do more than we have ever done before to identify it or, better yet, stop it.

Traditionally, in fraud cases, samples and models are used to identify customers that characterize a certain kind of profile. The problem with this is that although it works, we are profiling a segment and not the granularity at an individual transaction or person level. As per customer experiences, it is estimated that only 20 percent (or maybe less) of the available information that could be useful for fraud modeling is actually being used. The traditional approach is shown in the following Figure.

Finally, not only do we need to perform analysis on the longevity of the logs to determine trends and patterns and to find failures, but also we need to ensure the analysis is done on all the data.

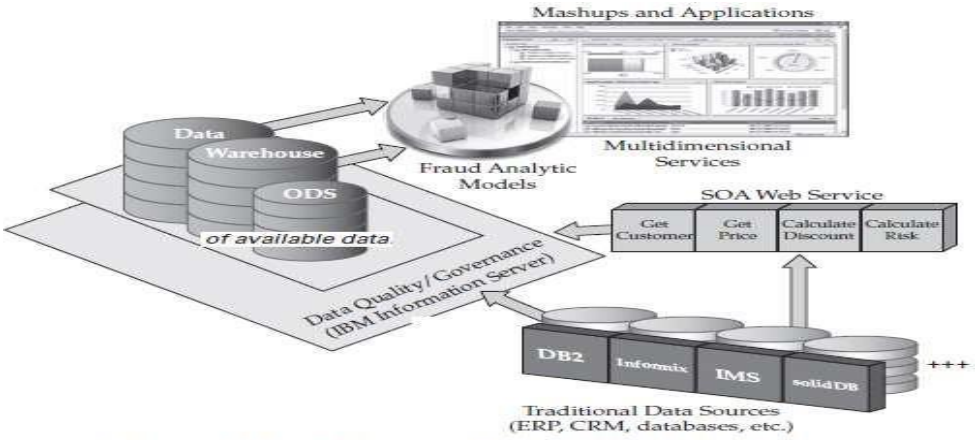
Log analytics is actually a pattern that IBM established after working with a number of companies, including some large financial services sector (FSS) companies. This use case comes up with quite a few customers since; for that reason, this pattern is called *IT for IT*. If we are new to this usage pattern and wondering just who is interested in IT for IT Big Data solutions, we should know that this is an internal use case within an organization itself. An internal IT for IT implementation is well suited for any organization with a large data center footprint, especially if it is relatively complex. For example, service-oriented architecture (SOA) applications with lots of moving parts, federated data centers, and so on, all suffer

from the same issues outlined in this section.

Some of large insurance and retail clients need to know the answers to such questions as, “What are the precursors to failures?”, “How are these systems all related?”, and more. These are the types of questions that conventional monitoring doesn‟t answer; a Big Data platform finally offers the opportunity to get some new and better insights into the problems at hand.

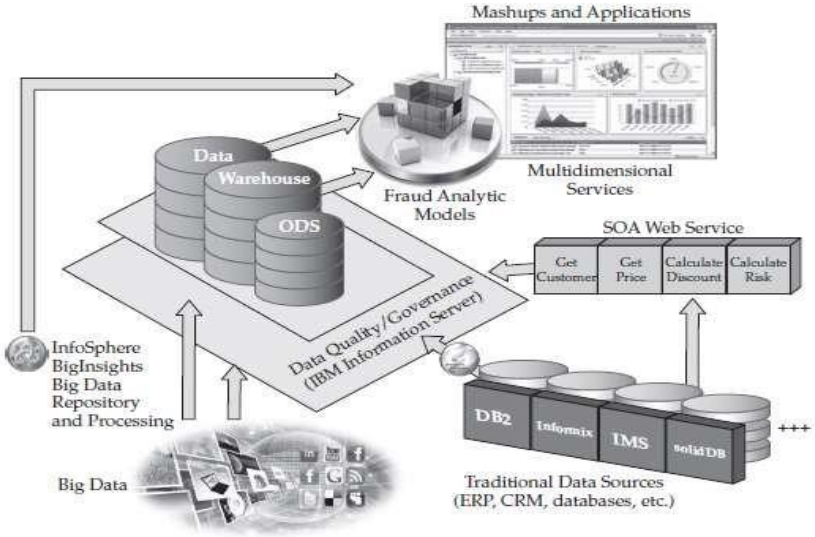
4. The Fraud Detection Pattern: Fraud detection comes up a lot in the financial services vertical, we will find it in any sort of claims- or transaction-based environment (online auctions, insurance claims, underwriting entities, and so on). Pretty much anywhere some sortof financial transaction is involved presents a potential for misuse and the universal threat of fraud. If we influence a Big Data platform, we have the opportunity to do more than we have ever done before to identify it or, better yet, stop it.

Traditionally, in fraud cases, samples and models are used to identify customers that characterize a certain kind of profile. The problem with this is that although it works, we are profiling a segment and not the granularity at an individual transaction or person level. As per customer experiences, it is estimated that only 20 percent (or maybe less) of the available information that could be useful for fraud modeling is actually being used. The traditional approach is shown in the following Figure.



Traditional fraud detection patterns use approximately 20 percent of available data

We can use BigInsights to provide an flexible and cost-effective repository to establish what of the remaining 80 percent of the information is useful for fraud modeling, and then feed newly discovered high-value information back into the fraud model as shown in the following Figure.



A modern-day fraud detection ecosystem synergizes a Big Data platform with traditional processes

A modern-day fraud detection ecosystem provides a low-cost Big Data platform for exploratory modeling and discovery. Typically, fraud detection works after a transaction gets stored only to get pulled out of storage and analyzed; storing something to instantly pull it back out again feels like latency to us. With Streams, we can apply the fraud detection models as the transaction is happening.

5. The Social Media Pattern: Perhaps the most talked about Big Data usage pattern is social media and customer sentiment. More specifically, we can determine how sentiment is impacting sales, the effectiveness or receptiveness of marketing campaigns, the accuracy of marketing mix (product, price, promotion, and placement), and so on.

Social media analytics is a pretty hot topic, so hot in fact that IBM has built a solution specifically to accelerate our use of it: Cognos Consumer Insights (CCI). CCI can tell what people are saying, how topics are trending in social media, and all sorts of things that affect the business, all packed into a rich visualization engine.

6. The Call Center Mantra: "This Call May Be Recorded for Quality Assurance Purposes": It seems that when we want our call with a customer service representative (CSR) to be recorded for quality assurance purposes, it seems the *may* part never works in our favor. The challenge of call center efficiencies is somewhat similar to the fraud detection pattern.

Call centers of all kinds want to find better ways to process information to address what"s going on in the business with lower latency. This is a really interesting Big Data use case, because it uses analytics-in-motion and analytics-at-rest. Using in-motion analytics (Streams) means that we basically build our models and find out what"s interesting based upon the conversations that have been converted from voice to text or with voice analysis as

the call is happening. Using at-rest analytics (BigInsights), we build up these models and then promote them back into Streams to examine and analyze the calls that are actually happening in real time: it‟s truly a closed-loop feedback mechanism.

7. Risk: Patterns for Modeling and Management: Risk modeling and management is another big opportunity and common Big Data usage pattern. Risk modeling brings into focusa frequent question when it comes to the Big Data usage patterns, "How much of our data do we use in our modeling?" The financial crisis of 2008, the associated subprime loan crisis,and its outcome has made risk modeling and management a key area of focus for financial institutions.

Two problems are associated with this usage pattern: "How much of the data will we use for our model?" and "How can we keep up with the data‟s velocity?" The answer to the second question, unfortunately, is often, "We can‟t." Finally, consider that financial services trend to move their risk model and dashboards to inter-day positions rather than just close-of-day positions, and we can see yet another challenge that can‟t be solved with traditional systems alone. Another characteristic of today‟s financial markets is that there are massive trading volumes requires better model and manage risk.

8. Big Data and the Energy Sector: The energy sector provides many Big Data use case challenges in how to deal with the massive volumes of sensor data from remote installations. Many companies are using only a fraction of the data being collected, because they lack the infrastructure to store or analyze the available scale of data.

Vestas is primarily engaged in the development, manufacturing, sale, and maintenance of power systems that use wind energy to generate electricity through its wind turbines. Its product range includes land and offshore wind turbines. At the time of wrote this book, it had more than 43,000 wind turbines in 65 countries on 5 continents. Vestas used IBM BigInsights platform to achieve their vision is about the generation of clean energy.

Data in the Warehouse and Data in Hadoop:

Traditional warehouses are mostly ideal for analyzing structured data from various systems and producing insights with known and relatively stable measurements. On the other hand, Hadoop-based platform is well suited to deal with semi-structured and unstructured data, as well as when a data discovery process is needed.

The authors could say that data warehouse data is trusted enough to be "public," while Hadoop data isn‟t as trusted (*public* can mean vastly distributed within the company and not for external consumption), and although this will likely change in the future, today this is something that experience suggests characterizes these repositories.

Hadoop-based repository scheme stores the entire business entity and the reliability of

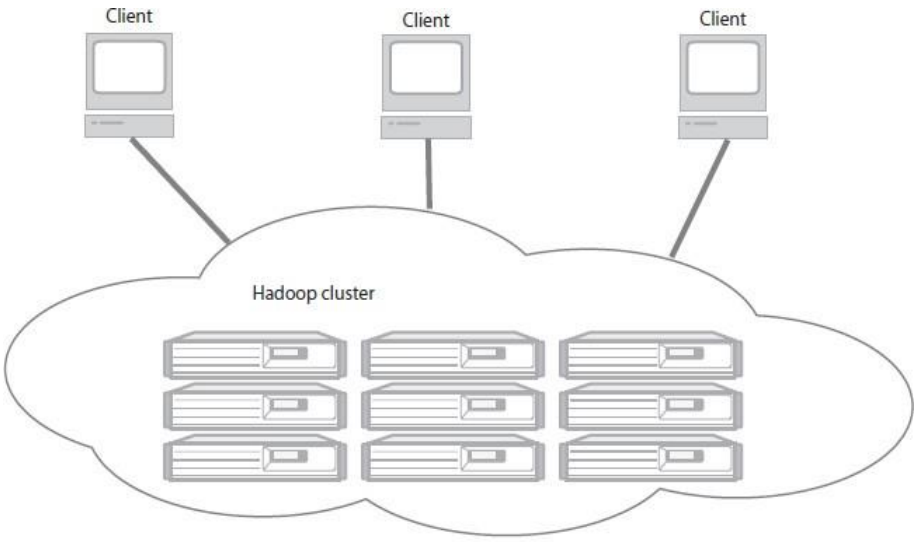
the Tweet, transaction, Facebook post, and more is kept intact. Data in Hadoop might seem of low value today. IT departments pick and choose high-valued data and put it through difficult cleansing and transformation processes because they know that data has a *high known value per byte*.

Unstructured data can‟t be easily stored in a warehouse. A Big Data platform can store all of the data in its native business object format and get value out of it through massive parallelism on readily available components.

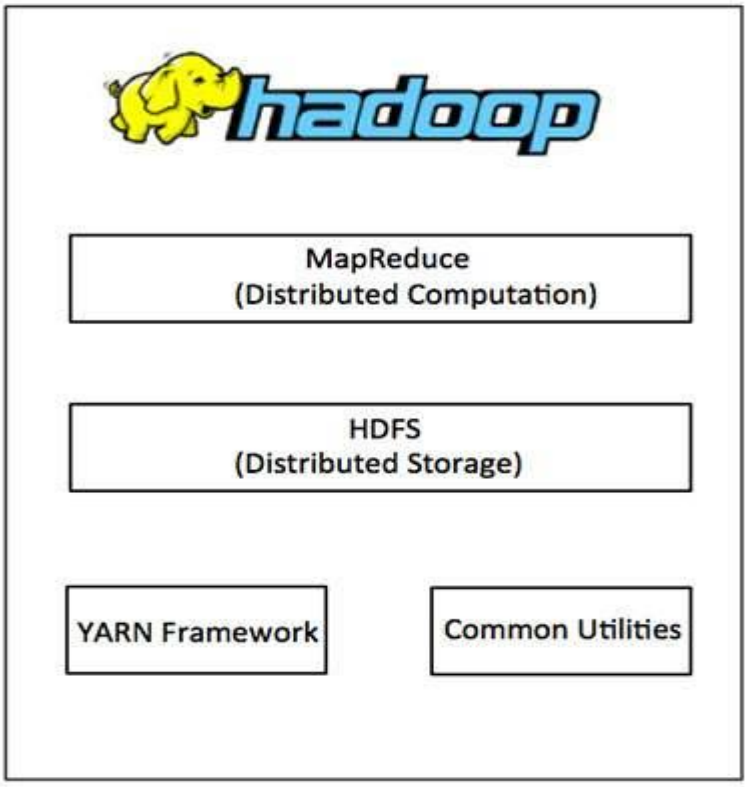
&&&&&&&&&

2. Introduction to Hadoop

Hadoop: Hadoop is an open source framework for writing and running distributed applications that process large amounts of data. Distributed computing is a wide and varied field, but the key distinctions of Hadoop are that it is: Accessible—Hadoop runs on large clusters of commodity machines or on cloudϖ computing services such as Amazon's Elastic Compute Cloud (EC2). Robust—Because it is intended to run on commodity hardware, Hadoop is architectedϖ with the assumption of frequent hardware malfunctions (errors). It can gracefully handle most such failures. Scalable—Hadoop scales linearly to handle larger data by adding more nodes to theϖ cluster. Simple—Hadoop allows users to quickly write efficient parallel code.ϖ The following Figure illustrates how one interacts with a Hadoop cluster. As we can see, a Hadoop cluster is a set of commodity machines networked together in one location. Data storage and processing all occur within this "cloud" of machines. Different users can submit computing "jobs" to Hadoop from individual clients, which can be their own desktop machines in remote locations from the Hadoop cluster.



A Hadoop cluster has many parallel machines that store and process large data sets. Client computers send jobs into this computer cloud and obtain results.



Understanding distributed systems and Hadoop: A lot of low-end/commodity machines tied together as a single functional is known as distributed system. A high-end machine with four I/O channels each having a throughput of 100 MB/sec will require three hours to read a 4 TB data set. With Hadoop, this same data set will be divided into smaller (typically 64 MB) blocks that are spread among many machines in the cluster via the Hadoop Distributed File System (HDFS). With a modest degree of replication, the cluster machines can read the data set in parallel and provide a much higher throughput. And such a cluster of commodity machines turns out to be cheaper than one highend server. Comparing SQL databases and Hadoop: SQL (structured query language) is designed for structured data. Many of Hadoop's initial applications deal with unstructured data such as text. From this perspective Hadoop provides a more general paradigm than SQL. SQL is a query language which can be implemented on top of Hadoop as the execution engine. But in practice, SQL databases tend to refer to a whole set of legacy technologies, with several dominant vendors, optimized for a historical set of applications. The following concepts explain a more detailed comparison of Hadoop with typical SQL databases on specific dimensions.

1. Scale-Out Instead of Scale-Up: Scaling commercial relational databases is expensive. Their design is friendlier to scaling up. To run a bigger database we need to buy a bigger machine which is expensive. Unfortunately, at some point there won't be a big enough machine available for the larger data sets. Hadoop is designed to be a scale-out architecture operating on a cluster of

commodity PC machines. Adding more resources means adding more machines to the Hadoop cluster. A Hadoop cluster with ten to hundreds of commodity machines is standard. In fact, other than for development purposes, there's no reason to run Hadoop on a single server.

2. Key/Value Pairs Instead of Relational Tables: A fundamental principle of relational databases is that data resides in tables having relational structure defined by a schema. Hadoop uses key/value pairs as its basic data unit, which is flexible enough to work with the less-structured data types. Hadoop, data can originate in any form (Structured/unstructured/ semi-structured), but it eventually transforms into (key/value) pairs for the processing functions to work on.

3. Functional Programming (Mapreduce) instead of Declarative Queries (Sql): SQL is fundamentally a high-level declarative language. By executing queries, the required data will be retrieved from database. Under MapReduce we specify the actual steps in processing the data, which is more similar to an execution plan for a SQL engine. Under SQL we have query statements; under MapReduce we have scripts and codes. MapReduce allows to process data in a more general fashion than SQL queries. For example, we can build complex statistical models from our data or reformat our image data. SQL is not well designed for such tasks.

4. Offline Batch Processing Instead of Online Transactions: Hadoop is designed for offline processing and analysis of large-scale data. It doesn't work for random reading and writing of a few records, which is the type of load for online transaction processing. In fact, Hadoop is best used as a write-once , read-many-times type of data store. In this aspect it's similar to data warehouses in the SQL world. Hadoop relates to distributed systems and SQL databases at a high level. Understanding MapReduce: MapReduce is a data processing model. The main advantage is easy scaling of data processing over multiple computing nodes. Under the MapReduce model, the data processing primitives are called mappers and reducers. Decomposing a data processing application into mappers and reducers is sometimes nontrivial. But, once we write an application in the MapReduce form, scaling the application to run over hundreds, thousands, or even tens of thousands of machines in a cluster is merely a configuration change. This is the reason what has attracted many programmers to the MapReduce model.

Ex: To count the number of times each word occurs in a set of documents. We have a set of documents having only one document with only one sentence:

Do as I say, not as I do.

We derive the word counts shown as the following.

Word	Count
as	2
do	2
i	2
not	1
say	1

When the set of documents is small, a straightforward program will do the job and pseudo-code is:

```
define wordCount as Multiset;
for each document in documentSet {
    T = tokenize(document);
    for each token in T {
        wordCount[token]++;
    }
}
display(wordCount);
```

The program loops through all the documents. For each document, the words are extracted one by one using a tokenization process. For each word, its corresponding entry in a multiset called wordCount is incremented by one. At the end, a display() function prints out all the entries in word Count.

The above code works fine until the set of documents we want to process becomes large. If it is large, to speed it up by rewriting the program so that it distributes the work over several machines. Each machine will process a distinct fraction of the documents. When all the machines have completed this, a second phase of processing will combine the result of all the machines. The pseudocode for the first phase, to be distributed over many machines, is

```
define wordCount as Multiset;
for each document in documentSubset {
    T = tokenize(document);
    for each token in T {
        wordCount[token]++;
    }
}
sendToSecondPhase(wordCount);
```

The pseudo-code for the second phase is:

```
define totalWordCount as Multiset;
for each wordCount received from firstPhase {
    multisetAdd (totalWordCount, wordCount);
}
```

This word counting program is getting complicated. To make it work across a cluster of distributed machines, we need to add a number of functionalities:

a. Store files over many processing machines (of phase one).
a. Write a disk-based hash table permitting processing without being limited by RAM capacity.
b. Partition the intermediate data (that is, wordCount) from phase one.
c. Shuffle the partitions to the appropriate machines in phase two.

Scaling the same program in MapReduce:

MapReduce programs are executed in two main phases, called *mapping* and *reducing*. Each phase is defined by a data processing function, and these functions are called *mapper* and

reducer, respectively. In the mapping phase, MapReduce takes the input data and feeds each data element to the mapper. In the reducing phase, the reducer processes all the outputs from the mapper and arrives at a final result. In simple terms, the mapper is meant to *filter and transform* the input into something that the reducer can *aggregate* over.

The MapReduce framework was designed in writing scalable, distributed programs. This two-phase design pattern is using in scaling many programs, and became the basis of the framework. Partitioning and shuffling are common design patterns along with mapping and reducing. The MapReduce framework provides a default implementation that works in most situations. MapReduce uses *lists* and *(key/value) pairs* as its main data primitives. The keys and values are often integers or strings but can also be dummy values to be ignored or complex object types. The map and reduce functions must obey the following constraint on the types of keys and values.

	Input	Output
map	<k1, v1>	list(<k2, v2>)
reduce	<k2, list(v2)>	list(<k3, v3>)

In the Map Reduce framework we write applications by specifying the mapper and reducer. The following steps explain the complete data flow:

1. The input to the application must be structured as a list of (key/value) pairs, list (<k_1, v_1>). The input format for processing multiple files is usually list (<String filename, String file_content>).
2. The list of (key/value) pairs is broken up and each individual (key/value) pair, <k_1, v_1>, is processed by calling the map function of the mapper. For word counting, mapper takes <String filename, String file_content> and promptly ignores filename. It can output a list of <String word, Integer count>, we can output a list of <String word, Integer 1> with repeated entries and let the complete aggregation be done later. That

is, in the output list we can have the (key/value) pair <"foo", 3> once or we can have the pair <"foo", 1> three times.

1. The output of all the mappers are (conceptually) aggregated into one giant list of $<k_2, v_2>$ pairs. All pairs sharing the same k2 are grouped together into a new (key/value) pair, $<k_2, list(v_2)>$. The framework asks the reducer to process each one of these aggregated (key/value) pairs individually. For example, the map output for one document may be a list with pair <"foo", 1> three times, and the map output for another document may be a list with pair <"foo", 1> twice. The aggregated pair the reducer will see is <"foo", list(1,1,1,1,1)>. In word counting, the output of our reducer is <"foo", 5>, which is the total number of times "foo" has occurred in our document set. Each reducer works on a different word. The Map Reduce framework automatically collects all the $<k_3, v_3>$ pairs and writes them to file(s).

Pseudo-code for map and reduce functions for word counting:

```
Map (String filename, String document)
    { List<String> T =
    tokenize(document);for each token
    in T {

    emit ((String)token, (Integer) 1);

    }
}

reduce (String token, List<Integer> values)
    {Integer sum = 0;

    for each value in values
    {sum = sum + value;
```

In the above pseudo-code, a special function is used in the framework called emit() which is used to generate the elements in the list one at a time. The emit() function further relieves the programmer from managing a large list. But Hadoop makes building scalable distributed programs easy.

Counting words with Hadoop—running your first program:

To run Hadoop on a single machine is mainly useful for development work. Linux is the official development and production platform for Hadoop, although Windows is a

supported development platform as well. For a Windows box, we'll need to install cygwin (http://www-cygwin.com/) to enable shell and Unix scripts.

To run Hadoop requires Java (version 1.6 or higher). Mac users should get it from Apple. We can download the latest JDK for other operating systems from Sun at http://java.sun.com/javase/downloads/index.jsp (or) www.oracle.com. Install it and remember the root of the Java installation, which we'll need later.

To install Hadoop, first get the latest version release at http://hadoop.apache.org/core/releases.html. After we unpack the distribution, edit the script "conf/Hadoop-env.sh" to set JAVA_HOME to the root of the Java installation we have remembered from earlier. For example, in Mac OS X, we'll replace this line

export JAVA_HOME=/usr/lib/j2sdk1.5-sun

with the following line

export JAVA_HOME=/Library/Java/Home

We'll be using the Hadoop script quite often. Run the following command:

bin/Hadoop

We only need to know that the command to run a (Java) Hadoop program is bin/hadoop jar <jar>. As the command implies, Hadoop programs written in Java are packaged in jar files for execution. The following command shows about a dozen example programs prepackaged with Hadoop:

bin/hadoop jar hadoop-*-examples.jar

One of the program is "wordcount". The important (inner) classes of that program are:

WordCount.java

```
public class WordCount extends Configured implements Tool {

  public static class MapClass extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable> {

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value,
                    OutputCollector<Text, IntWritable> output,
                    Reporter reporter) throws IOException {
      String line = value.toString();
      StringTokenizer itr = new StringTokenizer(line);          (1) Tokenize using white spaces
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());                              (2) Cast token into Text object
        output.collect(word, one);
      }
    }
  }

  public static class Reduce extends MapReduceBase
    implements Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterator<IntWritable> values,
                       OutputCollector<Text, IntWritable> output,
                       Reporter reporter) throws IOException {
      int sum = 0;
      while (values.hasNext()) {
        sum += values.next().get();
      }
      output.collect(key, new IntWritable(sum));                (3) Output count of each token
    }
  }

  ...
}
```

1. WordCount uses Java's StringTokenizer in its default setting, which tokenizes based only on whitespaces. To ignore standard punctuation marks, we add them to the StringTokenizer's list of delimiter characters:

   StringTokenizer itr = new StringTokenizer(line, " \t\n\r\f,.:;?![]'");

   When looping through the set of tokens, each token is extracted and cast into a Text Object.

2. In Hadoop, the special class Text is used in place of String. We want the word count to ignore capitalization, so we lowercase all the words before turning them into Text objects.

   word.set(itr.nextToken().toLowerCase());

   Finally, we want only words that appear more than four times.

3. We modify to collect the word count into the output only if that condition is met. (This is Hadoop's equivalent of the emit() function in our pseudo-code.)

   if (sum > 4) output.collect(key, new IntWritable(sum));

   After making changes to those three lines, we can recompile the program and execute it again. The results are shown in the following table.

Words with a count higher than 4 in the 2002 State of the Union Address

11th (5)	citizens (9)	its (6)	over (6)	to (123)
a (69)	congress (10)	jobs (11)	own (5)	together (5)
about (5)	corps (6)	join (7)	page (7)	tonight (5)
act (7)	country (10)	know (6)	people (12)	training (5)
afghanistan (10)	destruction (5)	last (6)	protect (5)	united (6)
all (10)	do (6)	lives (6)	regime (5)	us (6)
allies (8)	every (8)	long (5)	regimes (6)	want (5)
also (5)	evil (5)	make (7)	security (19)	war (12)
America (33)	for (27)	many (5)	september (5)	was (11)
American (15)	free (6)	more (11)	so (12)	we (76)
americans (8)	freedom (10)	most (5)	some (6)	we've (5)
				and so on

Without specifying any arguments, executing wordcount will show its usage information:

bin/hadoop jar hadoop-*-examples.jar wordcount

which shows the arguments list:

wordcount [-m <maps>] [-r <reduces>] <input> <output>

The only parameters are an input directory (<input>) of text documents we want to analyze and an output directory (<output>) where the program will dump its output. To execute wordcount, we need to first create an input directory:

mkdir input

and put some documents in it. We can add any text document to the directory. To see the wordcount results:

bin/hadoop jar hadoop-*-examples.jar wordcount input output more output/*

We'll see a word count of every word used in the document, listed in alphabetical order. The source code for wordcount is available and included in the installation at src/examples/org/apache/hadoop/examples/WordCount.java. We can modify it as per our requirements.

History of Hadoop:

Hadoop is a versatile (flexible) tool that allows new users to access the power of distributed computing. By using distributed storage and transferring code instead of data, Hadoop avoids the costly transmission step when working with large data sets. Moreover, the redundancy of data allows Hadoop to recover should a single node fail. It is easy of creating programs with Hadoop using the MapReduce framework. On a fully configured cluster, "running Hadoop" means running a set of daemons, or resident programs, on the different servers in the network. These daemons have specific roles; some exist only on one server, some exist across multiple servers. The daemons include:

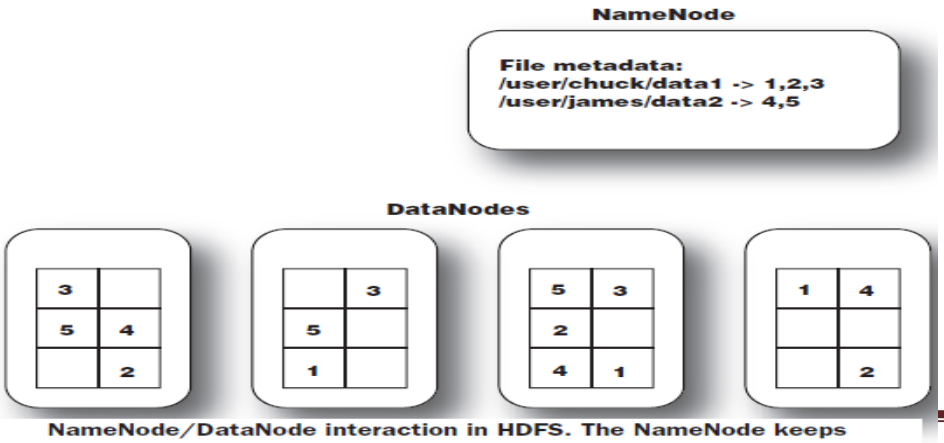
1. NameNode
2. DataNode
3. Secondary NameNode
4. JobTracker
5. TaskTracker

1. NameNode: The distributed storage system is called the *Hadoop File System*, or HDFS. The NameNode is the master of HDFS that directs the slave DataNode daemons to perform the low-level I/O tasks. The NameNode is the bookkeeper of HDFS; it keeps track of how the files are broken down into file blocks, which nodes store those blocks, and the overall health of the distributed filesystem.

The function of the NameNode is memory and I/O intensive. As such, the server hosting the NameNode typically doesn't store any user data or perform any computations fora MapReduce program to lower the workload on the machine. This means that the NameNode server doesn't double as a DataNode or a TaskTracker.

There is unfortunately a negative aspect to the importance of the NameNode—it's a single point of failure of the Hadoop cluster. For any of the other daemons, if their host nodes fail for software or hardware reasons, the Hadoop cluster will likely continue to function smoothly or we can quickly restart it and Not so for the NameNode.

2. DataNode: Each slave machine in the cluster will host a DataNode daemon to perform the grunt work of the distributed filesystem—reading and writing HDFS blocks to actual files on the local filesystem. When we want to read or write a HDFS file, the file is broken into blocks and the NameNode will tell the client which DataNode each block resides in. The client communicates directly with the DataNode daemons to process the local files corresponding to the blocks. Furthermore, a DataNode may communicate with other DataNodes to replicate its data blocks for redundancy. The following figure illustrates the roles of NameNode and DataNodes.



NameNode/DataNode interaction in HDFS. The NameNode keeps track of the file metadata—which files are in the system and how each file is broken down into blocks. The DataNodes provide backup store of the blocks and constantly report to the NameNode to keep the metadata current.

The data1 file takes up three blocks, which we denote 1, 2, and 3, and the data2 file consists of blocks 4 and 5. The content of the files are distributed among the DataNodes. In this illustration, each block has three replicas. For example, block 1 (used for data1) is replicated over the three rightmost DataNodes. This ensures that if any one DataNode crashesor becomes inaccessible over the network, we'll still be able to read the files.

DataNodes are constantly reporting to the NameNode. Each of the DataNodes informs the NameNode of the blocks it's currently storing. After this mapping is complete, the DataNodes continually poll the NameNode to provide information regarding local changes as well as receive instructions to create, move, or delete blocks from the local disk.

3. Secondary NameNode: The Secondary NameNode (SNN) is an assistant daemon for monitoring the state of the cluster HDFS. Like the NameNode, each cluster has one SNN, and it typically resides on its own machine as well. No other DataNode or TaskTracker daemons run on the same server. The SNN differs from the NameNode in that this process doesn't receive or record any real-time changes to HDFS. Instead, it communicates with the NameNode to take snapshots of the HDFS metadata at intervals defined by the cluster configuration.

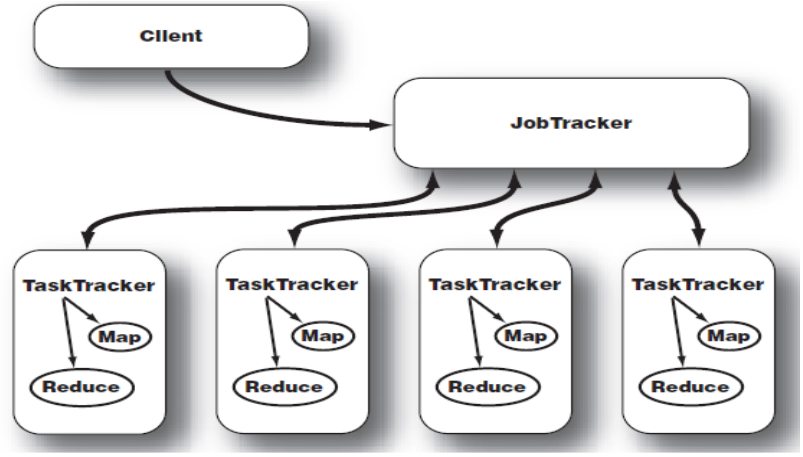
The NameNode is a single point of failure for a Hadoop cluster, and the SNNsnapshots help minimize the downtime and loss of data. However, a NameNode failure requires human intervention to reconfigure the cluster to use the SNN as the primary NameNode.

4. JobTracker: There is only one JobTracker daemon per Hadoop cluster. It's typically run on a server as a master node of the cluster. The JobTracker daemon is the link between our application and Hadoop. Once we submit our code to the cluster, the JobTracker determines the execution plan by determining which files to process, assigns nodes to different tasks, and monitors all tasks as they're running. If a task fails, the JobTracker will automatically relaunch the task, possibly on a different node, up to a predefined limit of retries.

5. TaskTracker: The JobTracker is the master control for overall execution of a MapReduce job and the TaskTrackers manage the execution of individual tasks on each slave node. The interaction between JobTracker and TaskTracker is shown in the following diagram.

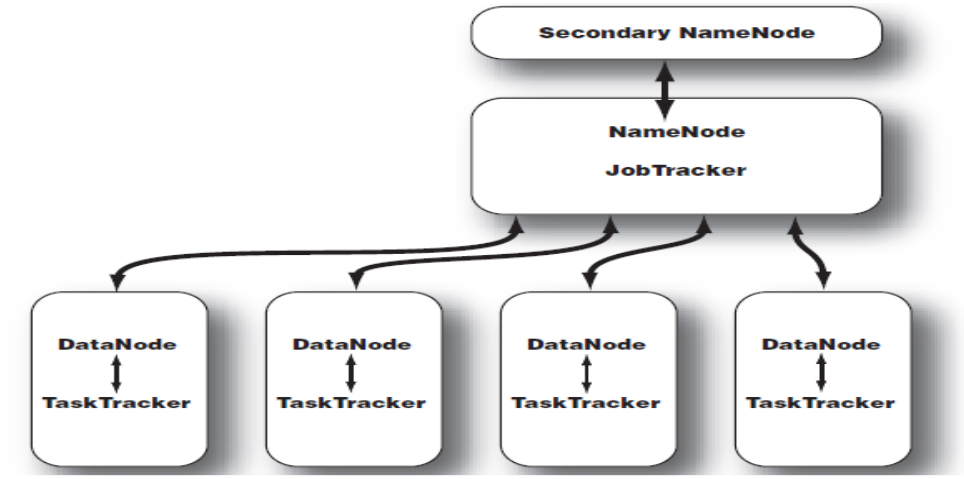
Each TaskTracker is responsible for executing the individual tasks that the JobTracker assigns. Although there is a single TaskTracker per slave node, each TaskTracker can spawn multiple JVMs to handle many maps or reduce tasks in parallel.

One responsibility of the TaskTracker is to constantly communicate with the JobTracker. If the JobTracker fails to receive a heartbeat from a TaskTracker within aspecified amount of time, it will assume the TaskTracker has crashed and will resubmit the corresponding tasks to other nodes in the cluster.



JobTracker and TaskTracker interaction. After a client calls the JobTracker to begin a data processing job, the JobTracker partitions the work and assigns different map and reduce tasks to each TaskTracker in the cluster

The topology of one typical Hadoop cluster in described in the following figure:



Topology of a typical Hadoop cluster. It's a master/slave architecture in which the NameNode and JobTracker are masters and the DataNodes and TaskTrackers are slaves.